UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

João Pedro Costa Barnabé

Códigos Polares Aplicados em Canais de Múltiplo Acesso

Campinas

2020

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

João Pedro Costa Barnabé

Códigos Polares Aplicados em Canais de Múltiplo Acesso

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Telecomunicações e Telemática.

Orientador: Prof. Dr. Gustavo Fraidenraich

Este exemplar corresponde à versão final da dissertação defendida pelo aluno João Pedro Costa Barnabé, e orientada pelo Prof. Dr. Gustavo Fraidenraich

_____

Campinas

2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

Barnabé, João Pedro Costa, 1993-
B252c Códigos polares aplicados em canais de múltiplo acesso / João Pedro Costa Barnabé. – Campinas, SP : [s.n.], 2020.

Orientador: Gustavo Fraidenraich.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Códigos corretores de erros (Teoria da informação). 2. Capacidade de canal. I. Fraidenraich, Gustavo, 1975-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Polar codes applied in multiple access channels
Palavras-chave em inglês:
Error correcting codes (Information Theory)
Channel capacity
Área de concentração: Telecomunicações e Telemática
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Gustavo Fraidenraich [Orientador]
Fabbryccio Akkazzha Machado Cardoso
Renato da Rocha Lopes
Data de defesa: 27-01-2020
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-7543-547X
- Currículo Lattes do autor: http://lattes.cnpq.br/9322490694591410

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: João Pedro Costa Barnabé | RA: 117406
Data da Defesa: 27 de janeiro de 2020

Título da Dissertação: "Códigos Polares Aplicados em Canais de Múltiplo Acesso".

Prof. Dr. Gustavo Fraidenraich (Presidente, FEEC/UNICAMP)
Dr. Fabbryccio Akkazzha Machado Cardoso (Membro Titular, CPqD)
Prof. Dr. Renato da Rocha Lopes (Membro Titular, FEEC/UNICAMP)

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Tese) e na secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

Dedico esta dissertação à minha família.

Agradecimentos

Gostaria de agradecer aos meus pais e minha família, que me deram todo o amor e carinho e me forneceram todas as chances e oportunidades necessárias para que eu pudesse chegar até aqui.

Sou grato a minha noiva, Arantxa, por me apoiar durante esses anos, compreender os dias ocupados e as noites que passei em claro durante este projeto e sempre me manter incentivado a trabalhar neste projeto e no nosso futuro.

Sou grato ao CPQD, principalmente meus chefes e colegas mais próximos, Dr. Fabbryccio Cardoso, Andre Luiz Nunes de Souza, Glauco Simões e Raoni Alcântara por me acompanharem na etapa inicial desta pesquisa e também por permitirem e viabilizarem a realização deste curso de Mestrado.

Agradeço aos meus colegas do laboratório Wisstek, por estarem ao meu lado durante todo o processo e me guiarem sobre os procedimentos padrão de uma pesquisa, incluindo o de escrita deste trabalho.

Por fim, agradeço também ao meu orientador, o professor Dr. Gustavo Fraidenraich, que, além de sempre estar disposto a me ouvir e me orientar a qual caminho seguir, também me propôs esta oportunidade de pesquisa, além de muitas outras, que abriu este mundo para mim.

O presente trabalho foi realizado com apoio do CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil.

"Talento é um interesse que se busca. Qualquer coisa que você esteja disposto a treinar, você consegue fazer."
(Robert Norman)

Resumo

Este trabalho propõe-se o uso de Códigos Polares no Canal de Múltiplo Acesso (MAC) com dois usuários transmitindo e um usuário como receptor. O esquema proposto utiliza a modulação *Binary Phase Shift Keying* (BPSK), e o receptor utiliza a técnica de Cancelamento Sucessivo de Interferência (SIC). Os resultados mostram-se promissores já que, usando um tamanho de palavra código de $N = 4096$, obteve-se um par de taxas a menos de 3 dB e $0.32\,\frac{bits/s}{Hz}$ da região de capacidade.

O trabalho também analisa o desempenho dos Códigos Polares em um canal ponto a ponto com entrada discreta e saída contínua com as as seguintes técnicas de decodificação: Cancelamento Sucessivo (SC); Cancelamento Sucessivo em Lista (SCL); SCL associado ao uso de CRC (*cyclic redundancy check*); e decodificação em lista adaptativa.

Palavras-chaves: Códigos Polares; Canal de Múltiplo Acesso; Capacidade de Canal; Codificação de Canal.

Abstract

This work proposes the use of Polar Codes for the Multiple Access Channel (MAC) two transmitting users and one user as receiver. The proposed scheme uses Binary Phase Shift Keying (BPSK) modulation, and the receiver uses the Successive Interference Cancellation (SIC) technique. The results are promising since using a codeword size of $N = 4096$ yields a pair of rates less than 3 dB and 0.32 $\frac{bits/s}{Hz}$ from the capacity region.

The paper also analyzes the performance of Polar Codes in a discrete input and continuous output point-to-point channel with the following decoding techniques: Successive Cancellation (SC); Successive List Cancellation (SCL); SCL associated with the use of CRC (cyclic redundancy check); and adaptive list decoding

Keywords: Polar Codes; Multiple Access Channel; Channel Capacity; Channel Coding.

Lista de ilustrações

Figura 1 – Diagrama de blocos de um sistema de comunicação. 16
Figura 2 – Canal de múltiplo acesso com dois usuários. 23
Figura 3 – Região de capacidade para o canal de múltiplo acesso com dois usuários e distribuição de entrada fixa. 24
Figura 4 – Diagrama que demonstra a codificação sistemática de Polar Codes para N = 8 e K = 4. 30
Figura 5 – Diagrama de codificação para N = 8 e K = 5. 31
Figura 6 – Diagrama de decodificação para N = 8. 31
Figura 7 – BER mostrado por Arikan para *Polar Codes* sistemático com N = 256, R = 0.5 e SNR de design = 0dB comparado com *Polar Codes* tradicionais sob as mesmas condições. 32
Figura 8 – Modelo do Canal de Múltiplo Acesso Gaussiano. 43
Figura 9 – PDF do sinal recebido (Y), com $\frac{E_b}{N_0} = 12dB$ e $R_1 = R_2 = 1$. . . 44
Figura 10 – PDF do sinal recebido para um canal ponto a ponto com $\frac{E_b}{N_0} = 12dB$. 45
Figura 11 – Região de capacidade para canal de múltiplo acesso com duas entradas Gaussianas. 47
Figura 12 – Resultados obtidos para os decodificadores SC e SCL iterativo (SCLA) 54
Figura 13 – BER e FER (WER) mostrado por Arikan para Polar Codes sistemático com N = 256, R = 0.5 e SNR de design = 0dB (esquerda) e Resultados do simulador sob as mesmas condições (direita). 56
Figura 14 – Região de Capacidade calculada para 5.5 dB e pontos onde os pares de taxas resultaram em comunicação confiável (BER $< 10^{-5}$) 58

Lista de tabelas

Tabela 1 – Exemplo de construção de código 28
Tabela 2 – Exemplo da BRev das posições 30
Tabela 3 – Tabela mostrando tamanho médio de lista para casos com tamanhos máximos de lista diferentes e $\frac{E_b}{N_0}$ diferentes. 55

Lista de Acrônimos e Abreviações

$\mathcal{A}$	Alfabeto
AWGN	Ruído Aditivo Gaussiano Branco (*Additive White Gaussian Noise*)
B	Matriz contendo os valores de bit nas posições respectivas da matriz LIK
BEC	*Binary Erasure Channel*
BER	*Bit Error Rate*
BPSK	*Binary Phase Shift Keying*
BRev	*Bit-Reversal Order*
BSC	*Binary Symmetric Channel*
$\mathcal{C}$	Capacidade do canal (em bits/s/Hz)
C_{soma}	Capacidade Soma
CDMA	*Code Division Multiple Access*
CRC	*Cyclic Redundancy Check*
E_b	Energia de bit
E_s	Energia de símbolo
FDMA	*Frequency Division Multiple* Access
FER (WER)	*Frame (Word) Error Rate*
$I(X;Y)$	Informação Mútua entre X e Y
K	Número de bits de informação *ou* tamanho da palavra antes da codificação em bits
$F \otimes G$	Produto de Kronecker entre a matriz F e G
L	Tamanho da lista utilizada no decodificador SCL
LDPC	Low-Density Parity-Check Codes
LIK	Matriz contendo os valores de LLR
LLR	Log Likelihood Ratio
MAC	Canal de Múltiplo Acesso (*Multiple Access Channel*)
ML	Maximum Likelihood

N	Tamanho do bloco transmitido *ou* palavra codificada (em bits)
$\mathcal{N}$(média, variância)	Distribuição normal com determinada média e variância
N_0	Energia ou Variância do ruído
PDF	Função Densidade de Probabilidade (*Probability Density Function*)
R	Taxa de código
R_{max}	Taxa máxima
SC	Cancelamento Sucessivo (*Successive Cancellation* (método de decodificação de códigos polares))
SCL	Cancelamento Sucessivo em Lista (*Successive Cancellation List [Decoding]*)
SCLA	Cancelamento Sucessivo em Lista Adaptativo (*Successive Cancellation Adaptative List [Decoding]*)
SIC	Cancelamento Sucessivo de Interferência (*Successive Interference Cancellation*)
SNR	Razão Sinal Ruído (*Signal-to-Noise Ratio*)
TDMA	*Time Division Multiple Access*
V	Número de usuários, no caso de múltiplos usuários
X	Sinal transmitido
X_1	Sinal Transmitido pelo usuário 1
X_2	Sinal Transmitido pelo usuário 2
Y	Sinal recebido
Y_{new}	Sinal recebido após remover a estimação do sinal do primeiro usuário (caso MAC)
Z	Ruído
$\mathcal{Z}(W)$	Parâmetro de Bhattacharyya de um canal W

Sumário

1 Introdução 15
1.1 Comunicação Multiusuário 18
1.2 Motivação 19
1.3 Objetivos 19
2 Fundamentos 21
2.1 Códigos corretores de erro 21
2.2 Canais de Múltiplo Acesso (MAC) 23
3 Polar Codes 26
3.1 Fundamentos 26
3.2 Decodificador em Lista 37
3.3 Revisão dos Algoritmos para Polar Codes - Canal Ponto a Ponto . . 38
4 Canal de Múltiplo Acesso 43
4.1 Região de Capacidade 44
4.2 Cancelamento Sucessivo de Interferência 46
4.3 Revisão das Referências de MAC 48
5 Simulador 49
5.1 Simulação do Canal 51
5.2 Probabilidade de bit 51
5.3 Resultados 53
5.3.1 Canal Ponto a Ponto 53
5.3.1.1 Decodificadores Implementados no MATLAB . . . 53
5.3.1.2 Decodificador C++ 55
5.3.2 Testes para MAC 56
6 Conclusões 60
6.1 Trabalhos Futuros 61

Referências 62
ANEXO A Códigos utilizados 66

1 Introdução

Antes da década de 40, pensava-se que, garantidamente, ao aumentar a taxa de transmissão de informação sobre um determinado canal, a probabilidade de erro inevitavelmente aumentaria. O trabalho pioneiro de Shannon, em 1948 [1], provou que isto só é verdade para taxas acima da capacidade do canal. O trabalho também afirma que, certamente, existe alguma codificação que realiza uma transmissão confiável (sem erros) através do canal se, e somente se, a taxa de transmissão de informação for menor que a capacidade do canal. Mais do que isso, neste mesmo trabalho ele determina um valor mínimo de taxa para que a informação possa ser comprimida, a entropia. [1].

O trabalho de Shannon deu início à área chamada de teoria da informação, e revolucionou a área de telecomunicações. Enquanto a entropia serve como um limite de compressão dos dados, guiando as pesquisas no campo de codificação de fonte; a capacidade do canal fornece um limite teórico de desempenho para os códigos corretores de erro. Desde então, o foco das pesquisas na área de codificação de canal vem sendo desenvolver um código que alcance a capacidade, o limite de Shannon.

Os códigos polares, desenvolvidos por Arikan, comprovadamente alcançam o limite de Shannon [2] quando o tamanho de bloco vai para infinito. Arikan introduz em seu trabalho a teoria de polarização dos canais e descreve seu codificador e decodificador.

Os códigos polares são parte dos códigos de bloco, ou seja, as etapas de

codificação e decodificação podem ser realizada por uma multiplicação matricial. Neste código em específico, a matriz é de fácil construção, conforme mostrado no capítulo 3. Porém, uma das desvantagens deste código é que o valor de N (tamanho do bloco) deve ser uma potência de 2.

Considere um sistema de comunicação como na figura 1.

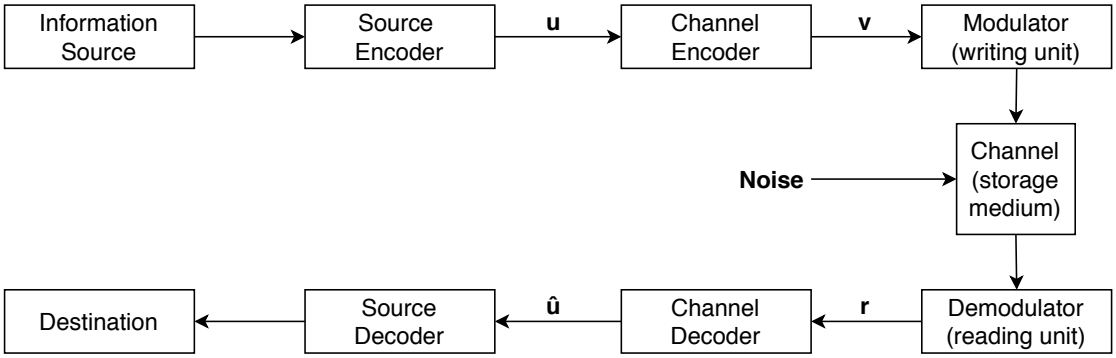

Figura 1 – Diagrama de blocos de um sistema de comunicação [3].

O primeiro bloco, codificador de fonte (*Source Encoder*), transforma a informação da fonte (seja ela digital ou analógica) em uma sequência de bits, que é aqui referida como o vetor de bits **u**. Além da conversão de analógico para digital (quando requerida), esse bloco também tem a responsabilidade de realizar a compressão da informação, de forma que o número de bits utilizado seja o mínimo possível e que a descompressão da palavra **u** possa ser (e seja) feita garantidamente sem ambiguidade.

O codificador de canal (Channel Encoder) é o bloco mais importante para este trabalho, em conjunto com o decodificador de canal. Sua função é adicionar redundância à palavra **u** de tal forma que proteja a palavra codificada contra o ruído ou outras interferências. No diagrama de blocos, essa palavra codificada é

referida como **v**. Também é considerado que a palavra codificada é discreta.

O próximo bloco, o modulador (*Modulator*), transforma os sinais discretos em sinais analógicos que possam ser transmitidos pelo canal. Como exemplo podemos ver o esquema de modulação utilizado neste trabalho, o BPSK (*Binary Phase Shift Keying*). Neste tipo de modulação um valor de tensão V é atribuído aos bits com valor 1 e $-V$ aos bits com valor 0. Adicionalmente, em um sistema de comunicação real, o modulador também transpõe o sinal a uma determinada frequência (frequência portadora) de forma que ele ocupe uma banda pré estabelecida independentemente do conteúdo do sinal.

Na saída do modulador, o sinal é enviado pelo canal, que introduz o ruído. No receptor, o primeiro bloco pelo qual o sinal passa é o demodulador (*Demodulator*). Este bloco traz o sinal de volta à frequência original (banda base) e processa o sinal recebido com o objetivo de prepará-lo a passar pelo resto do sistema. Este bloco transforma o sinal recebido em algo que o decodificador de canal consiga processar, seja isso valores de bits, ou até a probabilidade que cada bit tem de ser um determinado valor. Isto é enviado na forma de vetor **r**.

O próximo bloco, o decodificador de canal (*Channel Decoder*), transforma o vetor **r** de entrada em uma estimativa do vetor **u**, estimativa esta que chamamos de $\hat{\mathbf{u}}$, um vetor discreto.

Finalmente, o próximo (e último) bloco (decodificação de fonte, ou *Source Decoder*) tem como objetivo desfazer a compressão de dados do vetor $\hat{\mathbf{u}}$; para que o destinatário possa ter exatamente a mesma informação que a fonte desejava enviar.

Com ciência de como funciona um sistema de comunicação, pode-se falar sobre o foco do trabalho: um dos problemas abertos da área de telecomunicações é o de como alcançar a capacidade do canal de múltiplo usuários (MAC). Comumente, utiliza-se técnicas em que os usuários ou usam *time slots* diferentes (*Time-Division Multiple Access*) ou frequências diferentes (*Frequency-Division Multiple Access*). Contudo, estas técnicas não alcançam a capacidade do canal.

Este trabalho foca em técnicas de múltiplos usuários não-ortogonais (NOMA), onde existem vários usuários dividindo o mesmo canal, transmitindo sinais não ortogonais para um mesmo receptor. O foco deste trabalho é a utilização dos códigos polares neste canal e avaliação do seu desempenho.

1.1 Comunicação Multiusuário

A comunicação multi usuário é a extensão natural de uma comunicação ponto a ponto, que ocorre apenas entre dois usuários. Nos sistemas multi usuários, em particular, no canal de múltiplo acesso (MAC - *multiple access channel*), diversos usuários se comunicam com uma central (estação rádio base), semelhante ao *uplink* de um sistema celular.

Há diversas formas de comunicação multi usuários. Algumas utilizam a multiplexação na frequência (FDMA), outras utilizam a multiplexação no tempo (TDMA), neste trabalho no entanto, utilizaremos a mesma frequência e o mesmo *time slot*. A separação dos sinais será realizada pelo código corretor de erro, que utiliza distintos códigos para cada usuário.

1.2 Motivação

A principal motivação deste trabalho é avaliar o desempenho dos códigos polares em um canal de múltiplo acesso com dois usuários transmitindo e apenas um único usuário como receptor.

1.3 Objetivos

Os principais objetivos deste estudo são:

- Desenvolver um programa que simule, com sucesso, um sistema de comunicações de um canal de múltiplo acesso;
- Conseguir replicar o codificador e decodificador dos códigos polares;
- Aplicar técnicas já comprovadas que melhorem o desempenho dos códigos polares clássico;
- Comparar o desempenho dos códigos polares, aplicados a um canal de múltiplo acesso com ruído branco gaussiano aditivo, ao limite teórico de Shannon;

A organização deste trabalho foi realizada da seguinte forma:

- O Capitulo 2 apresenta uma fundamentação teórica sobre sistemas de comunicação, em particular sobre os códigos corretores de erro e canais de múltiplo acesso;

- O Capítulo 3 foca inteiramente nos códigos polares, principalmente nas técnicas que realmente foram utilizadas, além de conter uma revisão bibliográfica resumida mencionando as principais fontes de informação utilizadas;
- O Capítulo 4 contém todos os cálculos e as deduções realizadas em relação ao canal de múltiplo acesso para o caso específico deste trabalho;
- O Capítulo 5 contém todas as contribuições sobre o simulador, cálculo e desenvolvimento dos codificadores e decodificadores de canal;
- O Capítulo 6 contém os resultados comentados das simulações, tanto para o canal de um único usuário quanto para o canal MAC;
- Por fim, o capítulo 7 contém as conclusões do trabalho e possíveis trabalhos futuros.

2 Fundamentos

2.1 Códigos corretores de erro

Os códigos corretores de erro baseiam-se na ideia pioneira de Shannon [1] de que, usando um esquema de codificação e decodificação apropriado, pode-se reduzir (ou corrigir) os erros em uma mensagem sem reduzir a taxa de dados. [3]

Eles consistem, basicamente, nos blocos de codificação e decodificação de canal vistos na figura 1. Conforme mencionado, o processo de codificação adiciona redundância e a decodificação, consciente de como a codificação foi realizada, explora esta redundância para corrigir quaisquer erros que possam ter ocorrido na transmissão.

Um conceito importante neste ponto é o de taxa de código, definida como $R = \frac{K}{N}$. A taxa de código é a quantidade de bits de informação contidos em cada símbolo transmitido. A taxa também pode ser compreendida como o parâmetro que determina a quantidade de redundância adicionada na codificação; quanto menor o valor de R, maior a quantidade de redundância. A mensagem de tamanho K bits ingressa no codificador e sai com N bits, onde $N \geq K$ para modulação BPSK. A informação chega ao codificador com uma certa quantidade de energia que será utilizada na transmissão. Ao realizar a codificação, a energia dos K bits é dividida entre os N bits, de forma que quanto maior a taxa de código, menor a energia de cada bit transmitido.

A máxima taxa de código, para que haja uma comunicação confiável,

é chamada capacidade do canal. Considerando um canal $\mathcal{W} : \mathcal{X} \to \mathcal{Y}$, tem-se que sua capacidade é determinada por:

$$C = \max_{p(x)} I(X;Y) \ , \tag{2.1}$$

onde $I(X;Y)$ é a informação mútua entre X e Y e o máximo é obtido através de todas as possíveis distribuições de entrada $p(x)$ [3] [4].

Os códigos podem ser divididos em duas categorias: códigos convolucionais e códigos de bloco [3]. Os códigos convolucionais são códigos que contêm memória, ou seja, a saída do codificador não depende somente de sua entrada no momento, mas também de entradas anteriores. Um código convolucional de taxa $R = \frac{K}{N}$ e memória m pode ser feito com um circuito lógico sequencial com K bits de entrada, N bits de saída e os bits de entrada se mantém no codificador por m unidades de tempo. Mais detalhes podem ser encontrados em [3], contudo, os códigos polares, sobre os quais o trabalho fala, são códigos de bloco [2].

Nos códigos de bloco, o codificador divide a informação em blocos de tamanho K bits. Cada K-tupla binária é chamada de *mensagem* e representada como $\mathbf{u}= (u_1, u_2, u_3, ..., u_K)$; de tal forma que podem existir 2^K mensagens diferentes. Vale reforçar que, nos códigos de bloco, o valor K representa o tamanho de uma mensagem específica e não o comprimento de toda a informação a ser transmitida. O codificador passa o vetor $\mathbf{u}$ por uma transformada que resulta em uma N-tupla discreta $\mathbf{v}$ chamada de palavra código. Cada mensagem $\mathbf{u}$ é codificada independentemente uma da outra (ou seja, aqui, o codificador não possui memória), gerando assim várias palavras código também independentes uma das outras. Como para cada mensagem existe uma palavra código específica, pode-se assumir também que existem 2^K palavras código possíveis. O conjunto destas 2^K

palavras código de comprimento N é chamado de um código de bloco (N, K) [3].

2.2 Canais de Múltiplo Acesso (MAC)

O canal demostrado na figura 1 possui uma particularidade: assume-se que há somente um usuário transmitindo e um recebendo. Na prática, os canais possuem múltiplos usuários utilizando e compartilhando o mesmo canal. Um dos canais multi usuários é o canal de múltiplo acesso, onde existem dois ou mais usuários transmitindo informação e apenas um recebendo. Em casos assim, o receptor tem que lidar não somente com o ruído mas também com a interferência entre um sinal e outro [4].

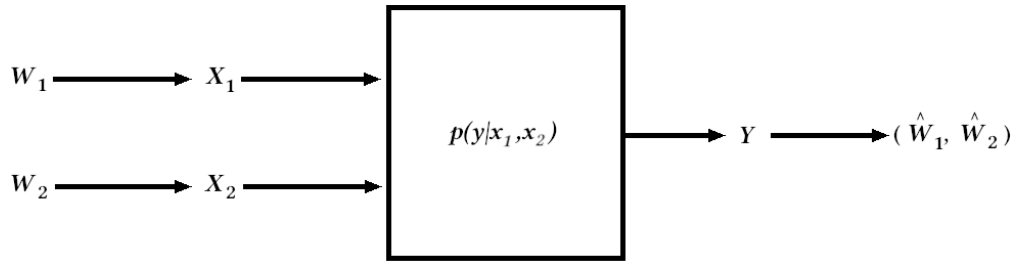

Figura 2 – Canal de múltiplo acesso com dois usuários. [4]

Considere que as taxas dos usuários 1 e 2 são R_1 e R_2, respectivamente. Assim como em um canal ponto a ponto, tem-se uma relação unidimensional onde o valor da capacidade do canal determina o valor máximo da taxa de código, demonstrada pela equação (2.1); para um canal de múltiplos usuários existe uma região V-dimensional (onde V é o número de usuários) que contém todas as V-tuplas de valores de taxa de código possíveis para os usuários. Para dois usuários $(V = 2)$, a região de capacidade se assemelha à da figura 3; onde cada ponto no

plano mostrado representa um par de taxas (R_1, R_2). Adicionalmente, os limites para as taxas R_1 e R_2 são definidos da seguinte forma [4]:

$$R_1 < I(X_1; Y|X_2)\,, \tag{2.2}$$

$$R_2 < I(X_2; Y|X_1)\,, \tag{2.3}$$

$$R_1 + R_2 < I(X_1, X_2; Y)\,. \tag{2.4}$$

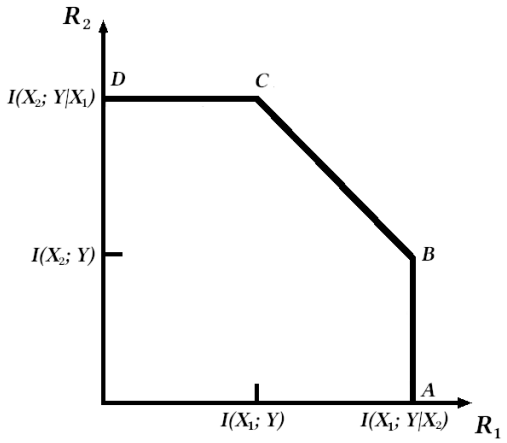

Figura 3 – Região de capacidade para o canal de múltiplo acesso com dois usuários e distribuição de entrada fixa [4].

Este trabalho foca em técnicas para lidar com o MAC em que os dois sinais sejam transmitidos ao mesmo tempo, na mesma frequência sem técnicas de divisão de múltiplo acesso (como, por exemplo, TDMA, FDMA ou CDMA).

Para alcançar a borda da região de capacidade de um canal MAC é necessário utilizar duas técnicas de decodificação: o cancelamento sucessivo de interferência (SIC) e o *time-sharing*.

A técnica SIC consiste em decodificar o sinal recebido como sendo o transmitido por um usuário, considerando o sinal do segundo usuário como interferência. Em seguida, com a estimação da primeira palavra em mãos, pode-se remover essa estimativa do sinal recebido com a finalidade de decodificar a mensagem do segundo usuário.

Referindo-se à figura 3, os retângulos formados pelos pontos $D, C, 0, I(X_1;Y)$ e $A, B, 0, I(X_2;Y)$ são as regiões contempladas por este método. Pode-se imaginar, por exemplo, que o ponto A representa a situação onde o usuário 2 não está transmitindo e o usuário 1 utiliza o canal solitariamente. Já o ponto B é o ponto máximo que pode-se elevar a taxa do usuário 2 para que ele possa ser decodificado corretamente na primeira decodificação enquanto o usuário 1 mantém a sua taxa.

O *time-sharing* consiste em alternar entre dois pares de taxa. O tempo que o codificador se mantém em cada par de taxa é o que determina em qual ponto da reta $\overline{CB}$ se está operando. Como exemplo, na figura 3, pode-se operar entre o ponto C com $\alpha\%$ do tempo e no ponto B com $(1-\alpha)\,\%$ do tempo. O valor de α definirá em que ponto da reta $\overline{CB}$ se está operando.

3 Polar Codes

3.1 Fundamentos

Polar Codes são baseados na concatenação de uma série de canais de tal forma que o canal final equivalente tenha uma capacidade 0 ou 1, o que é denominado polarização do canal. Ao final do processo a proporção de canais com capacidade 1, no caso da transmissão de uma palavra de tamanho infinito, é igual à capacidade do canal, alcançando, assim, o limite de Shannon (ao menos para canais binários, discretos, sem memória e com $N \to \infty$). Porém, como o tamanho de palavra transmitida é, em casos reais, limitado, o que se obtém é uma proporção de canais muito confiáveis e outra de canais pouco confiáveis, com relativamente poucos canais de confiabilidade média. O número de canais com capacidade ao redor de 0.5 fica menor à medida que o tamanho da palavra código N aumenta, o que também aumenta o número de canais com capacidade muito próximas de 0 ou 1 [2].

A definição central da polarização de canais é a seguinte: considere um canal binário discreto sem memória $\mathcal{W} : \mathcal{X} \to \mathcal{Y}$, sendo $\mathcal{W}$ o alfabeto de entrada $\{0, 1\}$ e $\mathcal{Y}$ o alfabeto de saída. As probabilidades de transição são definidas como $\mathcal{W}(y|x)$, onde $x \in \mathcal{X}$ e $y \in \mathcal{Y}$. Adicionalmente, $\mathcal{W}^N$ denota N usos de $\mathcal{W}$. Tem-se, portanto, $\mathcal{W}^N : \mathcal{X}^N \to \mathcal{Y}^N$ com $\mathcal{W}^N(y_1^N|x_1^N) = \prod_{i=1}^N \mathcal{W}(y_1|x_1)$. Para

$N = 2^n, n \geq 1$, considera-se:

$$X^N = U^N \cdot F^{\otimes n} \quad , \quad F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \tag{3.1}$$

onde $U^N = U_1, U_2, ..., U_N$, $X^N = X_1, X_2, ..., X_N$ e $F^{\otimes n}$ é a n-ésima potência de Kronecker da matriz F. Para qualquer $\delta > 0$ arbitrário e fixo é possível mostrar que (3.2),

$$\lim_{N\to\infty} \frac{|\{i \in [1, N]\} : \delta < H(U_i|U^{i-1}) < 1 - \delta|}{N} = 0. \text{ [2]} \tag{3.2}$$

É importante determinar quais serão os K bits mais confiáveis que serão utilizados como bits de informação. Os $N - K$ bits restantes serão denominados *frozen bits* e terão um valor fixo conhecido pelo decodificador. Essa etapa na construção do código é feita através da seleção dos canais que têm o menor valor de parâmetro de Bhattacharyya (limite máximo da probabilidade de erro de bloco ao usar o decodificador de *maximum likelihood* ML). O parâmetro de Bhattacharyya para um canal $\mathcal{W}$ é dado por $\mathcal{Z}(\mathcal{W}) \triangleq \sum_{y\in\mathcal{Y}} \sqrt{\mathcal{W}(y|0)\mathcal{W}(y|1)}$. A construção dos códigos polares através desse parâmetro dá-se ao escolher os canais com menor valor como canais de informação e o resto como bits congelados. Assume-se que há um vetor $z_N = \{z_{N,1}, z_{N,2}, \cdots, z_{N,N}\}$ que contém todos os N parâmetros de Bhattacharyya [2]. O cálculo segue a regra recursiva, mostrada mais explicitamente em [5]:

$$z_{2k,j} = \begin{cases} 2z_{k,j} - z_{k,j}^2 & \text{para } 1 \leq j \leq k, \\ z_{k,j-k}^2 & \text{para } k+1 \leq j \leq 2k, \end{cases} \tag{3.3}$$

Para $k = 1, 2, 2^2, \cdots, 2^{N-1}$, e $z_{1,1} = e^{-SNR_{design}}$, onde SNR_{design} é, idealmente, a relação sinal ruído (SNR) estimada, em seu valor linear.

Como exemplo foi realizada a construção de um código com as seguintes características: $SNR_{design} = 2dB$; $N = 8$ e $K = 4$. Os resultados obtidos estão presentes na tabela 1.

Posição	$z_{8,x}$	Tipo de bit	Ordem Crescente
1	0.8404	Frozen	8º
2	0.1578	Frozen	5º
3	0.2524	Frozen	6º
4	0.0035	Informação	2º
5	0.3606	Frozen	7º
6	0.0068	Informação	3º
7	0.0183	Informação	4º
8	$3.12 \cdot 10^{-6}$	Informação	1º

Tabela 1 – Exemplo de construção de código

Vale ressaltar que, teoricamente, esse método reduzido para o cálculo do parâmetro de Bhattacharyya só é válido para um canal BEC. Contudo, Arikan [2] mostra que pode ser usado em um canal BSC ou AWGN com uma perda desprezável de performance. Esta estratégia de construção acaba sendo mais atrativa por sua simplicidade, e normalmente acaba sendo a estratégia preferida visto que a técnica apropriada para estes canais é computacionalmente mais exigente [2].

Na codificação, é criado um vetor de tamanho N. Os K bits de informação são colocados nas posições confiáveis que são determinadas na construção do código. Já os bits restantes terão um valor fixo. Neste trabalho, todos os bits fixos foram definidos como 0, mas podem assumir qualquer valor fixo (desde que todos esses valores sejam conhecidos pelo decodificador). Arikan propõe codificar esse vetor de tamanho N com um código de bloco de taxa $R = 1$. A codificação

é realizada através da seguinte operação: $x_1^N = u_1^N \mathbf{G}_N$, onde $\mathbf{x}$ é a palavra codificada e $\mathbf{u}$ a concatenação dos bits de informação e bits congelados. A matriz $\mathbf{G}_N$ é construída a partir do produto de *Kronecker* da matriz F, que é definida da seguinte maneira (Lembrando que N é o tamanho da palavra após a codificação e $N = 2^n$) [2]:

$$\mathbf{G}_N = \mathbf{F}^{\otimes n}, \quad \mathbf{F} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \tag{3.4}$$

dessa forma:

$$\mathbf{x} = \mathbf{u} \cdot \mathbf{G}_N. \tag{3.5}$$

A operação realizada pela codificação combina N canais de tal forma a garantir a polarização. Percebe-se que a matriz G_N depende somente de N (ou n), portanto, após ser construída uma vez, não há necessidade de construí-la novamente até que se altere o tamanho do bloco.

A decodificação é feita usando o cancelamento sucessivo (SC). Este algoritmo calcula, passo a passo, a razão entre a probabilidade do bit ter valor 0 e a probabilidade do bit ter o valor 1 (*Log Likelihood Ratio*), dado os valores de todos os bits decodificados anteriormente. É importante ressaltar que a ordem em que esses bits são decodificados é a ordem inversa *bit reversed order* (*BRev*). A tabela 2 mostra um exemplo de decodificação para $N = 8$.

A figura 4 mostra um método usado para tornar os *Polar Codes* uma codificação sistemática. Esse método consiste em realizar o processo já descrito de codificação, mas, em seguida, passar os bits presentes nas K posições de informação pelo codificador novamente. Já a figura 7 mostra os benefícios na BER pelo uso

Posição	Binário	BRev	Ordem decodificação
0	000	000	1º
1	001	100	5º
2	010	010	3º
3	011	110	7º
4	100	001	2º
5	101	101	6º
6	110	011	4º
7	111	111	8º

Tabela 2 – Exemplo da BRev das posições

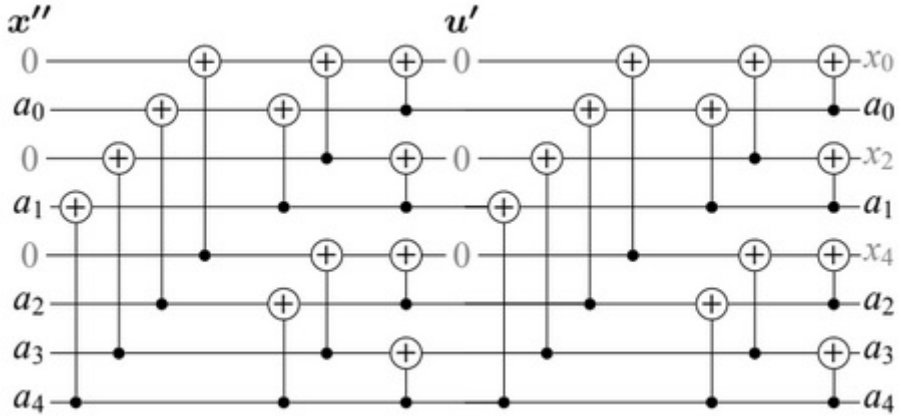

Figura 4 – Diagrama que demonstra a codificação sistemática de Polar Codes para N = 8 e K = 4 [6].

desse método. A justificativa fornecida por Arikan em [7] para a melhora é que a codificação sistemática aumenta a robustez do código à propagação de erro, devido ao fato de que os bits de informação agora podem ser diretamente observados pelo canal. Como o código apenas aumenta a robustez contra a propagação do erro, não a sua ocorrência, a melhora ocorre apenas na BER, já a FER não se altera entre as duas aplicações.

As figuras 5 e 6 mostram, respectivamente, para $N = 8$, o diagrama da codificação, e um diagrama simplificado da decodificação. Os números no diagrama

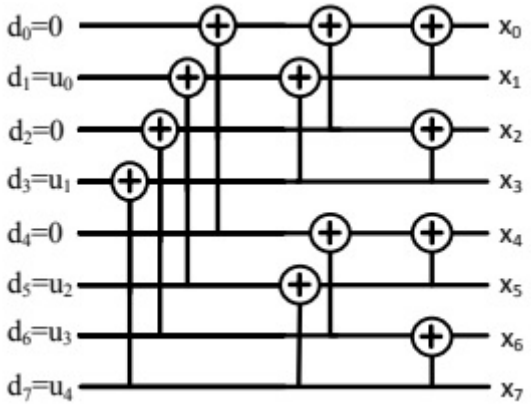

Figura 5 – Diagrama de codificação para N = 8 e K = 5 [5], na notação da imagem, o vetor **d** equivale ao vetor **u** deste trabalho.

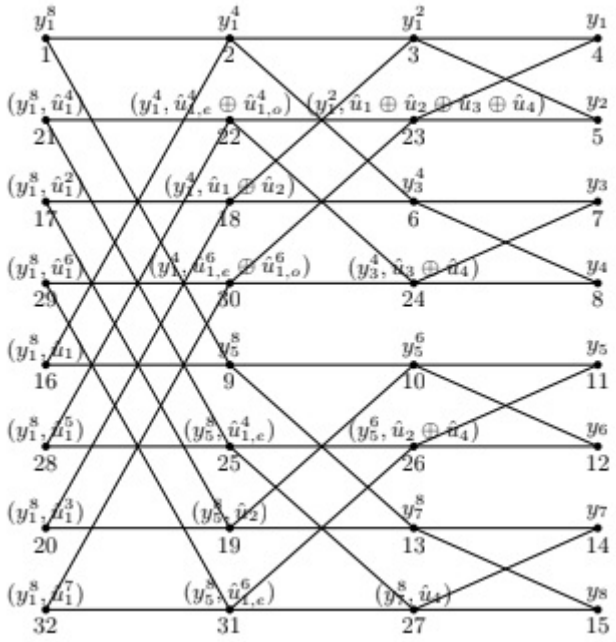

Figura 6 – Diagrama de decodificação para N = 8 [2].

de decodificação simbolizam a ordem que ela percorre (quais nós são ativados), enquanto as linhas indicam quais informações são necessárias para a obtenção da LLR do nó específico (quais são os próximos nós a serem ativados). Os nós à esquerda da linha necessitam das LLRs dos nós à direita. Os nós da decodificação

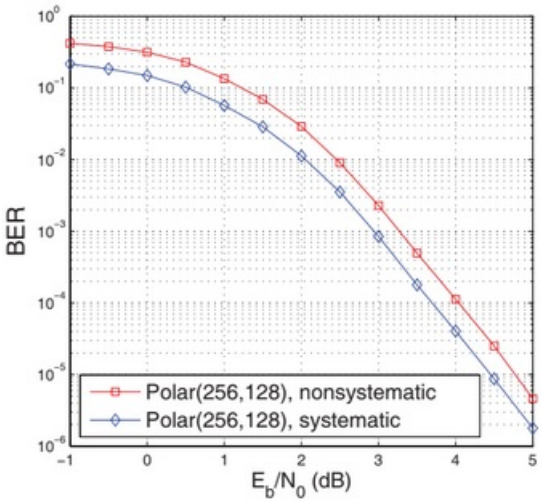

Figura 7 – BER mostrado por Arikan para Polar Codes sistemático com N = 256, R = 0.5 e SNR de design = 0dB comparado com *Polar Codes* tradicionais sob as mesmas condições [7].

são representados por uma matriz $LIK_{(n+1)\times N}$ cujos elementos são lik_{ij}, onde $2^n = N$. Além de LLR, cada nó possui um valor de bit, representados no código por uma matriz $B_{(n+1)\times N}$.

A decodificação SC estima $\hat{\mathbf{u}}$ através das LLRs de cada um dos bits. Na figura 6, a palavra recebida no receptor são os 8 bits na coluna mais à direita e a estimativa $\hat{\mathbf{u}}$ são representados pelos 8 bits da coluna mais à esquerda.

As LLRs da palavra recebida são utilizadas para calcular as LLRs de cada um dos nós e obter, ao final, as LLRs de $\hat{\mathbf{u}}$. Esse processo é dado pelas seguintes equações:

$$LLR_{nóhereup} = \text{sign}(LLR_{nóup}) \cdot \text{sign}(LLR_{nódown}) \cdot \min(|LLR_{nóup}|, |LLR_{nódown}|) \tag{3.6}$$

$$LLR_{nóheredown} = LLR_{nódown} - (2 \times B_{nóhereup} - 1) \cdot LLR_{nóup} \tag{3.7}$$

Como exemplo, pela figura 6, percebe-se que as conexões entre os nós são em formato de "x", tal que um par de nós em uma coluna está ligado a um par de nós em uma coluna adjacente, e essas conexões são exclusivas, ou seja, não há mais que uma conexão entre um par de nós. Os cálculos de LLR mencionados acima são feitos com 2 pares de nós que formam o "x". Por exemplo, os nós '1', '2', '16' e '9' seriam "nóhereup", "nóup", "nóheredown" e "nódown", respectivamente.

Quando obtém-se a LLR de um bit de $\hat{\mathbf{u}}$ (ou seja, um dos elementos da primeira coluna de LIK), é feita uma decisão do seu valor seguindo o critério (caso o bit não seja um dos *frozen bits*):

$$\hat{u}(i) = \begin{cases} 0, & se \quad lik_{i1} \geq 1 \\ 1, & se \quad não \end{cases} \tag{3.8}$$

Caso $\hat{u}(i)$ seja um *frozen bit*, assume-se o valor já conhecido previamente pelo decodificador, independentemente da LLR em seu nó.

Segue uma explicação resumida dos detalhes de como o decodificador funciona: o decodificador tenta primeiro descobrir a LLR do nó 1, esse cálculo, porém, necessita das LLRs dos nós 2 e 9. O código procede, então, de maneira recursiva, tentando calcular a LLR do nó 2 que, por sua vez, necessita das LLRs dos nós 3 e 6. O passo seguinte é calcular a LLR do nó 3, que necessita do valor dessa variável nos nós 4 e 5 que são, respectivamente, as LLRs dos bits 1 e 2 (ou 0 e 1 dependendo do tipo de nomenclatura) da palavra recebida (valor que é

conhecido). Com a LLR do nó 3 calculada, o próximo passo é calcular a LLR do nó 6 (que também depende da LLR de bits recebidos). Com os valores dos nós 3 e 6, pode-se calcular a LLR do nó 2. O código procede da mesma forma, calculando a LLR do nó 9 da mesma forma que fez com o nó 2 e, com essas duas informações, calculando a LLR do nó 1. Podendo, assim, tomar uma decisão sobre o valor do primeiro bit da palavra $\hat{\mathbf{u}}$.

O decodificador repete o procedimento acima para todos os bits. Vale lembrar que os valores das LLRs estão armazenados, ou seja, para calcular a LLR do nó 16 (que é o próximo passo) basta utilizar as LLRs dos nós 2 e 9 que foram calculadas anteriormente (além do valor do bit presente no nó 1, já que o nó 16 é o primeiro "nóheredown" que está sendo estimado).

Quando um bit de $\hat{\mathbf{u}}$ é decidido, seu valor é utilizado para se obter os valores dos bits da matriz B, seguindo o caminho visto no diagrama da codificação. Esses valores de B são utilizados para calcular alguns dos valores de LIK. O processo ocorre continuamente, bit a bit (seguindo a ordem do diagrama visto) até que se descubra todos os bits de $\hat{\mathbf{u}}$, terminando, assim, a decodificação SC.

Para o caso em que o método sistemático esteja sendo usado, a decodificação é feita de modo idêntico até esse ponto. Porém, referindo-se a figura 4, a palavra enviada é $\mathbf{x}$, e a palavra decodificada é uma estimação de $\mathbf{u}$. Como o código é sistemático, os bits de informação estão presentes na palavra transmitida na mesma localização original, ou seja, para obter a palavra desejada, codifica-se, com o codificador original (não sistemático) a palavra obtida pelo decodificador. Assim, a palavra resultante conterá os bits de informação pertinentes.

O decodificador em lista funciona de maneira similar ao SC, de tal

forma que se usarmos tamanho de lista igual a 1, o decodificador é o mesmo. Enquanto o decodificador SC decide se um bit de informação decodificado deve ser 1 ou 0 de acordo com o valor da LLR naquele ponto, o decodificador em lista assume que esses bits podem ser ou 1 ou 0, separando o caminho da decodificação em dois ramos diferentes para cada bit de informação decodificado (ao invés de decidir o valor no momento em que se obtém a LLR).

O tamanho da lista determina quantos desses caminhos podem ser armazenados, de forma que, quando se obtém um número maior de caminhos do que o tamanho da lista, os caminhos menos prováveis são eliminados.

O decodificador funciona da seguinte maneira: após a palavra codificada passar pelo canal e chegar no receptor, este calcula a probabilidade de cada um dos bits recebidos ser 1 e 0, criando assim dois vetores de tamanho N (um contendo as probabilidades dos bits assumirem o valor 1 e outro contendo a probabilidade de eles assumirem o valor 0). Em seguida, usando essas probabilidades, o decodificador calcula, sequencialmente, a probabilidade de cada um dos bits da informação pré-codificada ser 0 ou 1. Vale lembrar nesse ponto que a palavra que será decodificada irá conter N bits, com K bits de informação e N-K bits congelados (bits cujo valor é 0 e sabemos que é 0). Cada cálculo realizado utiliza os bits previamente calculados como condição, ou seja, considerando uma palavra de 8 bits onde todos são bits de informação.

Como exemplo, suponha que o vetor [a b c d e f g h] passou pela matriz codificadora, os cálculos ocorrem nessa ordem: $P(a = 1)\, e\, P(a = 0); P(b = 1|a = 1), P(b = 1|a = 0), P(b = 0|a = 1), P(b = 0|a = 0); P(c = 1|a = 0, b = 0), P(c = 1|a = 0, b = 1), P(c = 1|a = 1, b = 0), P(c = 1|a = 1, b = 1), P(c = 0|a = 0, b =$

$0), P(c = 0|a = 0, b = 1), P(c = 0|a = 1, b = 0), P(c = 0|a = 1, b = 1)$ e assim por diante.

Nesse exemplo, chega-se no ponto onde há 8 possíveis palavras. Caso o decodificador de lista usasse tamanho de lista 4, os 4 caminhos menos prováveis seriam descartados e não seriam considerados no cálculo das probabilidades do bit 'd'. Quando o bit decodificado for um dos bits congelados, o caminho que considera seu valor diferente do valor predeterminado é ignorado.

Depois de se obter as probabilidades dos L (tamanho da lista) caminhos mais prováveis, eles são ordenados por probabilidade. Assim como há um método para a obtenção da LLR dos nós, pode-se estimar a probabilidade de o valor do bit no nó ser 1 ou 0 usando um método semelhante. No SCL original, ao invés de considerar uma matriz de LLR (LIK), considera duas matrizes de probabilidade $W0_{(n+1)\times N}$ e $W1_{(n+1)\times N}$, onde $W0$ contém as probabilidades dos bits serem 0 e $W1$ de serem 1. Ao calcular as probabilidades de um bit de informação de $\hat{\mathbf{u}}$ a divisão de caminhos mencionada acima ocorre. Maiores detalhes de como o cálculo recursivo destas probabilidades é realizado são explicados no trabalho em [8].

Quando a divisão acontece, e o número de caminhos resultantes ultrapassa L, os valores $w0_{i1}$ e $w1_{i1}$ (onde i é o bit sendo decodificado) de todos os L caminhos são ordenados do maior para o menor. E os caminhos correspondentes aos menores valores são descartados.

Adicionalmente, esse artigo [8] também propõe que seja adicionado um CRC (*cyclic redundancy check*) à palavra a ser enviada antes da codificação. Esse CRC pode ser usado, no final da decodificação para saber se a palavra decodificada é uma das palavras possíveis. Ou seja, caso seja verificado, através do CRC, que a

palavra decodificada não é válida, ela não é escolhida, mesmo que a palavra esteja no topo da lista. O decodificador continua a verificar todas as palavras da lista, em ordem, até encontrar uma que passe no teste. Caso seja encontrada uma palavra desse modo, ela é escolhida como a saída do decodificador, caso não, é mantida a palavra no topo da lista mais próxima de correta.

Vale lembrar também que, para os *Polar Codes*, os valores de N e K são parametrizados no codificador e, portanto, podem ser alterados facilmente para se encaixar nas especificações desejadas. A única limitação é que N deve ser uma potência de 2. Esse limitação faz com que, por exemplo, não consigamos alcançar uma taxa de código de exatamente $\frac{1}{3}$, somente um valor próximo disso.

Arikan, em seu trabalho, realiza o BRev durante a codificação recursiva. Ele menciona a possibilidade de fazer a codificação por multiplicação matricial ou de realizar o BRev em qualquer ponto da codificação ou decodificação. No nosso trabalho o BRev é utilizado somente na ordem da decodificação pois se encaixou melhor no ambiente de programação do MATLAB. Na sessão de resultados pode-se perceber que essa escolha não causou nenhuma perda de desempenho.

3.2 Decodificador em Lista

Através de uma busca na Internet, foi encontrado um decodificador em lista implementado em c++ pela usuária "MashaYudi", do repositório *github*, encontrado em [9].

Como será visto na seção de resultados, o desempenho do decodificador em lista deixou a desejar no critério de velocidade. Para agilizar a simulação, o

decodificador foi implementado em c/c++, cuja execução é mais rápida. Para isso, utilizou-se o trabalho MashaYudi com algumas alterações no código em c++ e no script do MATLAB para que o uso de CRC fosse possível. Como o CRC foi aplicado em MATLAB, vale lembrar que, se o tamanho do CRC for maior que 0, isso fará com que a simulação demore ainda mais para ser executada (em alguns casos, o tempo chega a mais que dobrar). O algoritmo utilizado neste programa é basicamente uma tradução literal passo a passo dos algoritmos presentes em [8] para c++.

Todo o processo é dividido em 3 partes, cada uma com parâmetros que podem ser ajustados. Primeiramente, temos o construtor dos *Polar Codes*: aqui, pode-se escolher o tamanho da palavra codificada (N), o número de bits de informação (K), e a SNR para a qual o código será obtido (valor em dB). Em seguida, temos o codificador, que recebe como entrada a palavra a ser codificada, a posição dos bits a serem congelados, matriz de codificação (saídas do construtor), e os valores N e K; adicionalmente, recebe o tamanho do CRC. Por último, temos o decodificador, que recebe: a probabilidade dos bits serem 1, a probabilidade dos bits serem 0, os valores N e K, a posição dos bits congelados, o tamanho do CRC, o tipo de decodificação: (*hard decoding* ou *soft decoding*), e o tamanho da lista.

3.3 Revisão dos Algoritmos para Polar Codes - Canal Ponto a Ponto

A referência "www.polarcodes.com" [10] fornece uma biblioteca em MATLAB e também quais artigos foram utilizados para implementar esse código.

Os scripts dessa biblioteca permitem implementar a construção do código; implementar a codificação e decodificação dos Polar Codes. A decodificação é um pouco diferente da descrita no trabalho original de Arikan [2]). Esta biblioteca também permite implementar a codificação e decodificação usando *Systematic* Polar Codes [7] e também permite executar *scripts* para criar gráficos de desempenho.

Em [11], há um simulador do funcionamento dos *Polar Codes*. O primeiro simulador implementado em MATLAB foi baseado nesta referência. Enquanto Arikan sugere que façamos uma operação bit reversal para mudarmos a ordem dos bits após a codificação, o simulador de [11] faz isso a cada passo ao separar os bits ímpares dos pares.

O artigo original de Arikan [2] descreve o conceito de polarização de canal e dos códigos polares (*Polar Codes*). Em [7], Arikan propõe um método para tornar os *Polar Codes*, que é uma codificação inerentemente não sistemática, em sistemática. Os resultados desse trabalho mostram que isso elimina parcialmente a propagação de erros, melhorando a BER (*bit error rate*) mas, ao mesmo tempo, mantendo o FER (*frame error rate*) igual ao do método não sistemático.

Em [12], Arikan compara o desempenho dos *Polar Codes* com os códigos de Reed-Muller e comenta que os *Polar Codes* não precisam ser construídos em função do canal, mas isso beneficia muito o seu desempenho. Sugere também uma simplificação: ao invés de calcular o parâmetro de Bhattacharyya do canal específico do problema, que o cálculo seja feito assumindo um canal BEC equivalente (por exemplo, se o canal for BSC, calcular o parâmetro como se fosse um BEC onde a probabilidade de *erasure* seja igual a probabilidade de erro no BSC do problema).

Na sequência, há dois trabalhos dos mesmos autores do trabalho [11], que explicam algumas das alterações feitas nela em comparação ao sugerido originalmente em [2]. Um deles [13], discorre sobre a possibilidade de realizar a decodificação em ordem natural, ao invés de utilizar a ordem inversa (invertendo a ordem dos bits do número que designa a posição do canal, e ordenando os canais de acordo com esse novo índice), que é o sugerido em [2].

Em [5], são testados vários métodos para construção de *Polar Codes* para um mesmo canal AWGN. Entre eles, está o sugerido por Arikan em [12], usando o método original de [2] com a aproximação do parâmetro de Bhattacharyya para o cálculo recursivo. A conclusão que se chega é que, ao se assumir, para design do código, o SNR que maximiza o desempenho (que, para o mesmo caso, pode ser diferente para métodos de construção diferentes), os métodos têm desempenhos similares. Assim, pelo menos dentre os exemplos estudados em [5], é melhor escolher o método de construção mais simples, ou seja, o método de construção sugerido em [12].

Tanto a sua menção em [12] quanto os resultados comprovando sua eficácia em [5] mostram que a construção de código usando o cálculo aproximado do limite superior do erro é o mais interessante por sua simplicidade e eficácia. Sendo assim, será esse o método utilizado aqui.

Uma das melhorias feitas, de modo geral, aos *Polar Codes* é o método de decodificação por lista (SCL) proposto em [8]. Ele também segue a metodologia de SC mas tem um adicional. Ao invés de somente termos o caminho obtido ao seguir cegamente as *likelihood ratios* conforme a decodificação prossegue, esse método considera vários outros caminhos que não são ótimos *a priori*, mas podem trazer

um melhor resultado na decodificação.

Adicionalmente, [8] também propõe, no final, que seja adicionado um CRC (*cyclic redundancy check*) à palavra a ser enviada, antes da codificação. Esse CRC pode ser usado, no final da decodificação, para saber se a palavra decodificada é uma das palavras possíveis (o CRC pode ser comparado, nesse sentido à verificação de paridade). Ou seja, caso seja verificado, através do CRC, que a palavra decodificada não é válida, ela não é escolhida, mesmo sendo a palavra no topo da lista, então o decodificador continua a verificar todas a palavras da lista, em ordem, para achar a primeira que passe no teste. Caso nenhuma passe, opta-se pela que já estava no topo da lista.

O trabalho em [14] utiliza o CRC para agilizar o processo da decodificação em lista usando uma filosofia adaptativa e iterativa. O processo funciona da seguinte maneira: inicia-se a decodificação com tamanho de lista $L = 1$; caso a palavra decodificada passe no teste de CRC, ela é escolhida como saída, caso não, repete-se a decodificação com tamanho de lista $L = 2$. E esse ciclo é repetido, dobrando o tamanho de L a cada passo, até que se encontre uma palavra na lista que passe no CRC ou até que se alcance o tamanho máximo de lista, que é definido como parâmetro de entrada. Esse método de decodificação será referenciado como SCLA (*Successive Cancellation* em Lista, Adaptativo).

Outro fato interessante do decodificador SCLA é que, como a probabilidade de se decodificar corretamente uma palavra aumenta à medida que a SNR aumenta, nesses casos é mais provável a palavra passar pelo CRC com tamanhos de lista menor. Desta forma, o tempo de execução desse decodificador é inversamente proporcional à SNR.

Por último, percebeu-se que, originalmente, o método de lista decide e ordena os melhores caminhos de acordo com a probabilidade do bit ter o certo valor (0 ou 1). Contudo, pode ser interessante utilizar a LLR como parâmetro de entrada do decodificador, por questões particulares de implementação. Por este motivo, investigou-se o trabalho [15], onde os autores mostram que é possível utilizar a LLR como critério de ordenação de qualidade dos caminhos, após certas adaptações.

4 Canal de Múltiplo Acesso

O modelo de canal de múltiplo acesso é ilustrado na figura 8, onde X_1 e X_2 são as palavras codificadas pelos usuários 1 e 2, respectivamente; Z é o ruído branco gaussiano de variância N_0; e Y é o sinal que chega ao receptor. Dessa forma, pode-se descrever o sinal recebido como:

$$Y = X_1 + X_2 + Z \tag{4.1}$$

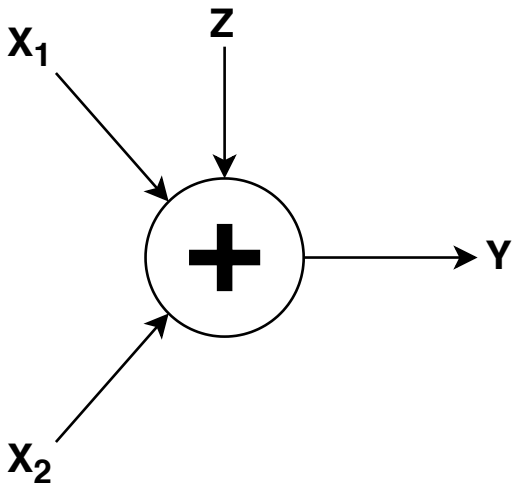

Figura 8 – Modelo do Canal de Múltiplo Acesso Gaussiano [4].

As palavras X_1 e X_2 fornecidas pelos usuários 1 e 2 são consideradas, aqui, variáveis aleatórias uniformes discretas independentes e identicamente distribuídas (*iid*), cujo alfabeto $\mathcal{A} = \{\sqrt{E_s}; -\sqrt{E_s}\}$. Desta forma, a PDF (função densidade de probabilidade) da amplitude do sinal recebido Y assume o formato

apresentado na figura 9 e é definido na equação (4.2).

$$p(y) = \sum_{x_1, x_2 \in \mathcal{A}} p(y|x_1, x_2) p(x_1, x_2)\,, \tag{4.2}$$

onde cada uma das probabilidades condicionais é uma gaussiana centrada em $x_1 + x_2$, ou seja:

$$p(y|x_1, x_2) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(y-(x_1+x_2))^2}{N_0}}\,. \tag{4.3}$$

Como X_1 e X_2 são *iid*, uniformes e a cardinalidade de $\mathcal{A}$ vale 2, i.e., $|\mathcal{A}| = 2$, temos que $p(x_1, x_2) = \frac{1}{4}$ independentemente dos valores de x_1 e x_2.

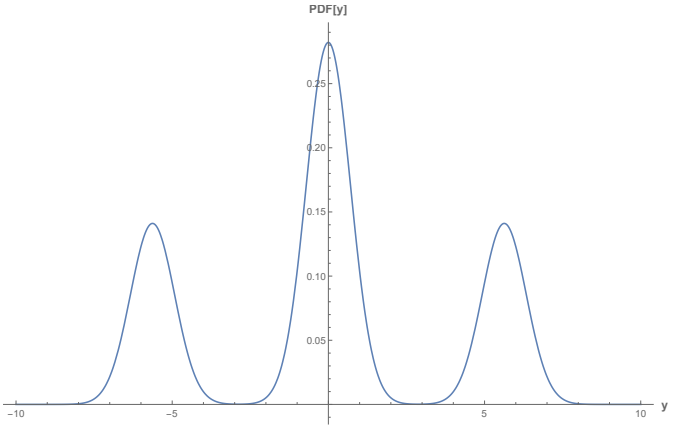

Figura 9 – PDF do sinal recebido (Y), com $\frac{E_b}{N_0} = 12dB$ e $R_1 = R_2 = 1$.

4.1 Região de Capacidade

Para formar a região de capacidade do canal, 3 inequações devem ser obedecidas, são elas as inequações (2.2), (2.3) e (2.4). Como os sinais dos usuários 1 e 2 são considerados independentes e identicamente distribuídos, o processo de

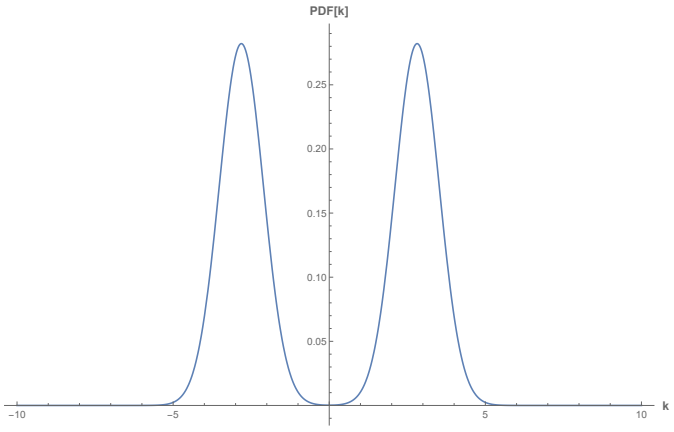

Figura 10 – PDF do sinal recebido para um canal ponto a ponto com $\frac{E_b}{N_0} = 12dB$.

obtenção dos valores máximos, $R_{1_{max}}$ e $R_{2_{max}}$ é idêntico para ambos. Dessa forma, será somente explicitado o procedimento de obtenção dos valores limites para as inequações (2.2), $R_{1_{max}}$, e (2.4), cujo valor máximo será chamado de C_{soma}.

A taxa máxima para o usuário 1, $R_{1_{max}}$, será calculada da seguinte forma:

$$R_{1_{max}} = I(X_1; Y|X_2)\,, \tag{4.4}$$

A informação mútua $I(X_1; Y|X_2)$ pode ser calculada usando o resultado dado em [16] para um canal ponto a ponto com entrada discreta e saída contínua dado por:

$$R_{1_{max}} = C = \sum_{x_1 \in \mathcal{A}} p(x_1) \int_{-\infty}^{\infty} p_{1u}(y|x_1) \log_2 \frac{p_{1u}(y|x_1)}{p_{1u}(y)} dy\,, \tag{4.5}$$

onde $p_{1u}(y|x_1)$ e $p_{1u}(y)$ são dadas por:

$$p_{1u}(y) = \sum_{x_1 \in \mathcal{A}} p_{1u}(y|x_1)p(x_1)\,, \tag{4.6}$$

$$p_{1u}(y|x_1) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(y-x_1)^2}{N_0}}\,. \tag{4.7}$$

A capacidade soma será calculada da seguinte forma:

$$R_1 + R_2 < C_{soma} = \sum_{x_1,x_2 \in A} p(x_1,x_2) \int_{-\infty}^{\infty} p(y|x_1,x_2) \log_2 \frac{p(y|x_1,x_2)}{p(y)} dy\,, \tag{4.8}$$

onde as PDFs $p(y|x_1,x_2)$ e $p(y)$ são dadas em (4.3) e (4.2), respectivamente.

Como exemplo, segue na figura 11 um caso contemplado no livro [4] onde os dois usuários transmitem sinais com PDFs gaussianas, o que simplifica as contas e resulta na região de capacidade mostrada. Esta figura também ilustra os resultados que podem ser idealmente alcançados por métodos ortogonais (como FDMA), representado pela linha curva no gráfico, enquanto a linha tracejada representa os resultados que podem ser obtidos ao se usar exclusivamente *time-sharing*.

4.2 Cancelamento Sucessivo de Interferência

O método de cancelamento sucessivo de interferência (SIC), mencionado nas sessões anteriores, consiste em alguns passos simples:

1. Decodificar o sinal recebido $Y = X_1 + X_2 + Z$ considerando X_2 como ruído e X_1 a palavra que se quer decodificar;

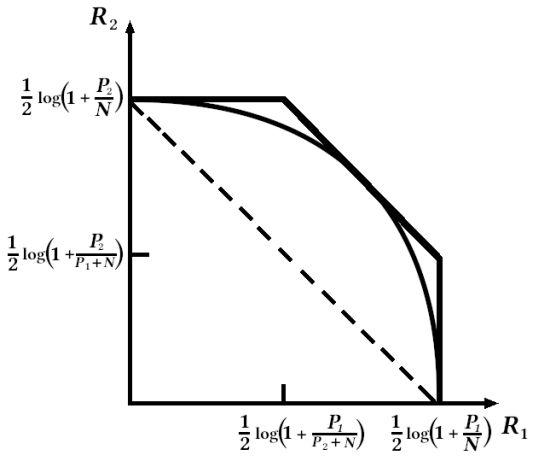

Figura 11 – Região de capacidade para canal de múltiplo acesso com duas entradas Gaussianas [4].

2. Após obter uma estimação $\hat{X}_1$ para X_1, subtrair esse valor em Y, De forma que $Y_{new} = Y - X_1$;

3. Decodificar Y_{new} a fim de obter-se X_2.

A ideia por trás deste método é remover a estimação de X_1 de Y, limpando o canal e ficando apenas X_2 e o ruído

$$X_1 \approx \hat{X}_1, \tag{4.9}$$

$$\therefore Y - \hat{X}_1 \approx X_2 + Z. \tag{4.10}$$

Tenha em mente que X_1 e X_2 podem ser invertidos nos passos acima. No entanto, nos testes realizados, a primeira palavra a ser decodificada foi a transmitida com a menor taxa.

4.3 Revisão das Referências de MAC

O livro escrito por Cover e Thomas [4] forneceu o conhecimento necessário tanto sobre o básico sobre o MAC, como o seu funcionamento e a sua capacidade para o canal com sinal gaussiano; quanto sobre também o método SIC, mencionando a ideia de funcionamento por trás dele. A teoria contida no livro foi a base para o desenvolvimento desta parte do trabalho.

Adicionalmente, alguns outros artigos forneceram alguns dados sobre desempenho de códigos corretores de erro no sistema MAC, estes artigos são [17], [18] e [19]. Sendo que [19] inclusive mostra o desempenho dos códigos polares neste âmbito, especificamente.

5 Simulador

Para o processo de simulação foi utilizado o método *Monte Carlo*. Simulam-se transmissões para cada um dos valores de E_b/N_0 desejados até que ou se obtenha 100 erros de bloco, ou que se chegue ao número máximo de transmissões por ponto (valor predefinido como entrada).

O simulador foi feito para o canal AWGN e modulação BPSK no MATLAB. O decodificador foi implementado em linguagem c++ e foi compilado com um compilador do próprio MATLAB. O algoritmo do simulador é o mostrado no algoritmo 1.

O código está como anexo em seu formato original, com extensão .m e .cpp (apenas para o decodificador). Para as simulações feitas com canal ponto a ponto, pode-se considerar o mesmo algoritmo, porém com $R_1 = 0$. Algumas partes do código, como a construção do código, codificação e decodificação em lista já possuem uma descrição completa de como funcionam em seus trabalhos originais ([8], [2]). Por este motivo, este trabalho não se focará em descrever, nos mínimos detalhes, os algoritmos específicos para esses blocos.

Os elementos do vetor $\mathbf{r}_x$ serão chamados de bits, apesar de serem números reais.

Algoritmo 1 Simulador

$frozenbits_1 = Polar\_Code\_Construction(N, K_1, E_B/N_0Design)$
$frozenbits_2 = Polar\_Code\_Construction(N, K_2, E_B/N_0Design)$
$E_B/N_0 = \{E_B/N_{0min}, \cdots, E_B/N_{0max}\}$
$i = 1;$
$R_1 = \frac{K_1}{N}$
$R_2 = \frac{K_2}{N}$
top:
$\frac{E_b}{N_0} = E_B/N_0(i)$
$E_{s_1} = 2 \cdot \frac{E_b}{N_0} \cdot R_1$
$E_{s_2} = 2 \cdot \frac{E_b}{N_0} \cdot R_2$
loop:
for $j = 1, 2$ **do**
    $u_j = rand[K]$
    $EncU_j = Codificador(u_j, frozenbits_j)$
    $x_j[N] = E_{s_j}(2EncU_j - 1)$
$t_x[N] = x_1 + x_2$
$r_x[N] = t_x + (Noise[N] \sim \ N(0, 1))$
$[W_{0_{in}}[N], W_{1_{in}}[N]] = Calc\_W_{first}(r_x, \frac{E_b}{N_0}, R_1, R_2)$
$\hat{u}_1 = Decodificador(W_{0_{in}}[N], W_{1_{in}}[N], frozenbits_1)$
$genie = Codificador(\hat{u}_1, frozenbits_1)$
$r_{new}[N] = r_x - E_{s_1}(2 \cdot genie - 1)$
$[W_{0_{in}}[N], W_{1_{in}}[N]] = Calc\_W(r_{new}, \frac{E_b}{N_0}, R_2)$
$\hat{u}_2 = Decodificador(W_{0_{in}}[N], W_{1_{in}}, frozenbits_2)$
$Contar\_Erros\_Bits(u_1, u_2, \hat{u}_1, \hat{u}_2)$
$Contar\_Erros\_Palavras(u_1, u_2, \hat{u}_1, \hat{u}_2)$
if *(ErrosPalavras*$\_x_1 \geq 100$ ***and*** *ErrosPalavras*$\_x_2 \geq 100$) **then**
    $Calc\_BER(Erros\_Bits)$
    if $\frac{E_b}{N_0} == E_B/N_{0max}$ **then**
        goto *end.*
    i ++
    goto *top.*
else
    if $Num_{iteracoes} \geq It_{max}$ **then**
        $Calc\_BER(Erros\_Bits)$
        goto *end.*
    goto *loop.*
end:
$Plot(BER\_x_1)$
$Plot(BER\_x_2)$

5.1 Simulação do Canal

Para que o simulador possa trabalhar de uma forma mais simplificada, o parâmetro de entrada referente à energia do sinal será a relação sinal ruído $\frac{E_b}{N_0}$. Infelizmente, dessa forma, não há como determinar valores separados para E_s e N_0, o que é necessário, já que o sinal recebido quando os bits transmitidos são '0' ou '1' possui distribuição normal $\mathcal{N}(-\sqrt{E_s}, \frac{N_0}{2})$ e $\mathcal{N}(\sqrt{E_s}, \frac{N_0}{2})$, respectivamente; onde $E_s = E_b \cdot R$.

Para contornar esta questão, será visto que a probabilidade de erro de bit só depende da razão $\frac{E_b}{N_0}$, e portanto pode-se fazer a transformação abaixo

$$\mathcal{N}(\pm\sqrt{E_s}, \frac{N_0}{2}) \equiv \mathcal{N}(\pm\sqrt{\frac{2E_s}{N_0}}, 1). \tag{5.1}$$

sem que haja mudança na probabilidade de erro de bit.

5.2 Probabilidade de bit

Para ambas as funções de cálculo de probabilidade de bit, o fator $\frac{E_b}{N_0}$ é um parâmetro de entrada, ou seja, assume-se que existe um estimador ótimo de canal presente no sistema.

Os trabalhos com canal AWGN [10] usam o valor da PDF, dada na equação (5.1), no ponto referente ao valor recebido, como sendo a probabilidade de um valor corresponder a um bit 0 ou 1. Como exemplo, considere o vetor de probabilidade dos bits assumirem valor igual a 0, $W_{0_{in}}$. Para obtê-lo, utiliza-se a seguinte equação:

$$W_{0_{in}}(j) = P\left(t_x(j) = 0 | r_x(j)\right), \tag{5.2}$$

onde $t_x(j)$ é a j-ésima posição do vetor de bits transmitido e $r_x(j)$ a j-ésima posição do vetor de bits recebido.

Considere também uma variável aleatória $\mathcal{D} \sim \mathcal{N}(-\sqrt{\frac{E_s}{N_0}}, 1)$ como sendo a variável do valor que um bit pode assumir receptor quando o bit '0' for transmitido. Assume-se na literatura que:

$$P\left(t_x(j) = 0 | r_x(j)\right) = f_{\mathcal{D}}(r_x(j)), \tag{5.3}$$

para $W_{1_{in}}$ faz-se o equivalente com $P\left(t_x(j) = 1 | r_x(j)\right)$ e uma variável aleatória com a distribuição adequada ($\mathcal{N}(\sqrt{\frac{E_s}{N_0}}, 1)$).

Contudo, para o canal de múltiplo acesso, os métodos descritos acima não estavam gerando os resultados esperados. A razão disso é que o sinal a ser decodificado na técnica SIC, ou seja, de um canal MAC, não possui a mesma PDF utilizada para o sinal de um canal ponto a ponto. As figuras 9 e 10 mostram a PDF do sinal recebido para o caso MAC e canal ponto a ponto. Assim, modificou-se a função que estima as probabilidades considerando as seguintes variáveis aleatórias: $\mathcal{D}_1 \sim \mathcal{N}(-\sqrt{\frac{E_{s_1}+E_{s_2}}{N_0}}, 1)$, $\mathcal{D}_2 \sim \mathcal{N}(-\sqrt{\frac{E_{s_1}-E_{s_2}}{N_0}}, 1)$, $\mathcal{S}_1 \sim \mathcal{N}(\sqrt{\frac{E_{s_1}+E_{s_2}}{N_0}}, 1)$ e $\mathcal{S}_2 \sim \mathcal{N}(\sqrt{\frac{E_{s_1}-E_{s_2}}{N_0}}, 1)$. As probabilidades são calculadas segundo as equações:

$$P\left(t_{x1}(j) = 0 | r_x(j)\right) = 0.5 \cdot \left(f_{\mathcal{D}_1}(r_x(j)) + f_{\mathcal{D}_2}(r_x(j))\right), \tag{5.4}$$

$$P\left(t_{x1}(j) = 1 | r_x(j)\right) = 0.5 \cdot \left(f_{\mathcal{S}_1}(r_x(j)) + f_{\mathcal{S}_2}(r_x(j))\right). \tag{5.5}$$

5.3 Resultados

5.3.1 Canal Ponto a Ponto

A seguir serão exibidos os resultados obtidos para a probabilidade de erro de bit para o canal com um único usuário, ou canal ponto a ponto.

5.3.1.1 Decodificadores Implementados no MATLAB

A figura 12 mostra a probabilidade de erro de bit versus a razão E_b/N_0 assumindo diferentes tamanhos de palavra N e diversos métodos de decodificação. Os limites de baixa e alta confiabilidade mostrados na imagem foram valores fornecidos pelo CPqD e eles se referem a valores utilizados como referência para canais de voz e dados, respectivamente.

No decodificador SC, percebe-se que seu desempenho melhora com o aumento de N, o que é previsto na teoria (o limite de Shannon será alcançado quando $N = \infty$). Também como previsto a BER cai à medida que $\frac{E_b}{N_0}$ aumenta. Além disso, o decodificador em lista iterativo apresenta um desempenho melhor do que o decodificador convencional. Para $N = 1024$, há um ganho de aproximadamente 1 dB entre o decodificador SC e o SCL iterativo com Lmax=16. Para $N = 4096$, há um ganho de aproximadamente 0.8 dB entre o decodificador SC e o SCL iterativo com $L_{max} = 16$.

Infelizmente, as vantagens do decodificador SCL não vêm sem o custo do aumento do tempo de execução. Enquanto o decodificador SC opera com complexidade $O(N \log N)$, o SCL opera com $O(L \times N \log N)$. Ou seja, o tamanho da lista multiplica o tempo de execução.

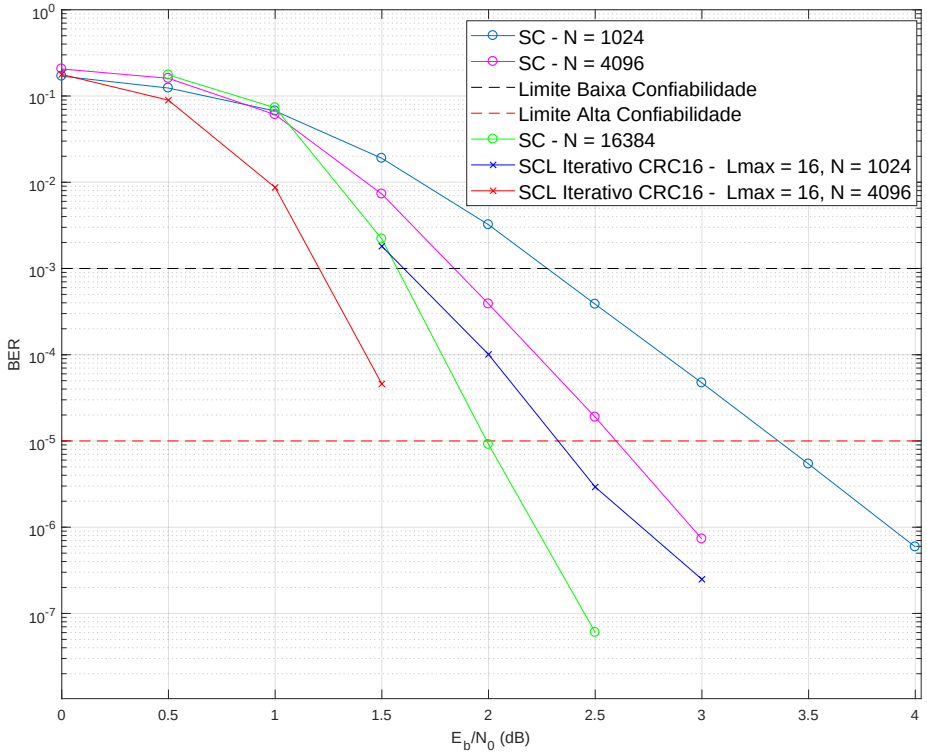

Figura 12 – Resultados obtidos para os decodificadores SC e SCL iterativo (SCLA)

O SCLA melhora esse tempo pois o tamanho de lista efetivamente usado, L_{ad} é bem menor do que o tamanho de lista máximo. A tabela 3 mostra os resultados em [14], e fornece qual seria o tamanho de lista equivalente variando a razão E_b/N_0 e também o L_{max}.

Nota-se portanto que a complexidade do algoritmo SCLA é $O(L_{ad} \times N \log N)$.

E_b/N_0 (dB)	1.0	1.2	1.4	1.6	1.8	2.0
$L_{max} = 32$	16.64	8.03	3.86	2.04	1.39	1.14
$L_{max} = 128$	35.31	12.16	4.52	2.17	1.41	
$L_{max} = 512$	70.41	19.14	5.45	2.27		
$L_{max} = 2048$	133.40	30.80	6.64	2.36		
$L_{max} = 8192$	271.07	52.59	7.88	2.47		

Tabela 3 – Tabela mostrando tamanho médio de lista para casos com tamanhos máximos de lista diferentes e $\frac{E_b}{N_0}$ diferentes [14].

5.3.1.2 Decodificador C++

O simulador com o decodificador em C++ de "MashaYudi" adaptado teve um desempenho bem melhor. A adição do algoritmo de CRC, porém, faz o tempo de execução aumentar e varia com alguns parâmetros tais como tamanho da palavra, tamanho da lista, e SNR.

Na figura 13 tem-se uma comparação do simulador usando tamanho de lista 1 sem CRC com os resultados do artigo de Arikan sobre Polar Codes sistemático [7]. Vemos resultados semelhantes, o que significa que, pelo menos na condição de lista tamanho 1 e sem o CRC, o algoritmo funciona perfeitamente.

No canal de múltiplo acesso é necessário trabalhar com taxas de código variadas. Por conta disso, foram usados os resultados presentes em [20], onde há exemplos de BER (WER) para códigos polares com decodificação SC com taxa $R \approx \frac{1}{3}$ e $R \approx \frac{2}{3}$ para testar o desempenho do sistema ponto a ponto de códigos polares para taxas diferentes de $R = \frac{1}{2}$. Após confirmar que o código funciona também para estas situações, os testes para MAC se iniciaram.

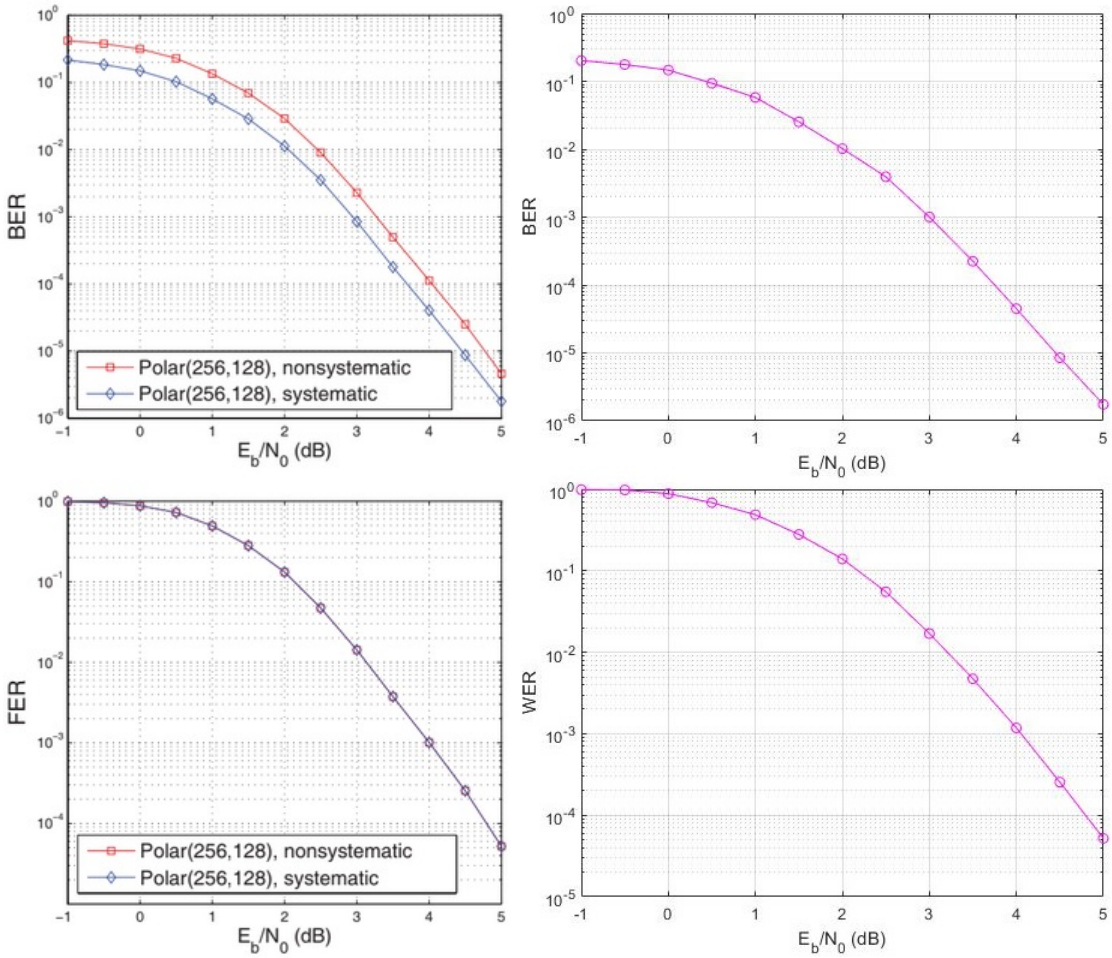

Figura 13 – BER e FER (WER) mostrado por Arikan para Polar Codes sistemático com N = 256, R = 0.5 e SNR de design = 0dB (esquerda) [7] e Resultados do simulador sob as mesmas condições (direita).

5.3.2 Testes para MAC

Sobre o método de avaliação das técnicas utilizadas, dois pontos foram escolhidos:

1. Calcular a região de capacidade para um determinado valor de $\frac{E_b}{N_0}$ (no caso, o valor escolhido foi $5.5dB$) e verificar se de fato, apresentam taxa de erro

de bit de no máximo, $10^{-}5$ neste valor de $\frac{E_b}{N_0}$;

2. Escolhe-se o melhor par de taxas (R_{1best}, R_{2best}) que atendem ao primeiro teste, e verifica-se a distância desse ponto ao limite de Shannon de duas maneiras distintas: 1) verificando qual $\frac{E_b}{N_0}$ produz a capacidade soma $C_{soma} = R_{1best}+R_{2best}$;2) verificando a diferença entre o valor de $\frac{E_b}{N_0}$ teórico e simulado.

Os testes foram realizados nas seguintes condições:

- $N_1 = N_2 = 4096$;
- R, e portanto K, variáveis para ambos os usuários;
- $\frac{E_b}{N_0} = 5.5dB$;
- Simulador realizava transmissões até haver 100 erros de palavra ou até 100 mil realizações (testes anteriores mostraram que uma taxa de erro de palavra igual a 10^{-3} corresponde a, aproximadamente, uma taxa de erro de bit de 10^{-5}, que é o valor desejado);
- Tamanho de lista $L = 4$;
- Como mencionado anteriormente, sem CRC;
- SNR_{design} = -2 dB.

O valor escolhido para SNR_{design} foi escolhido pois verificou-se que este valor levava a um melhor desempenho para ambos usuários através de tentativa e erro. Este valor pode ser otimizado para cada usuário e pode-se reduzir este valor na construção do código para o usuário com a menor taxa.

A figura 14 mostra os pares de taxa com taxa de erro de bit abaixo de 10^{-5} em $\frac{E_b}{N_0} = 5.5dB$. Os pontos destacados (os maiores) no gráfico mostram os pares de taxa com maior taxa soma nessa região. Como exemplo, o par $(0.75, 0.35)$, leva a uma taxa soma de $1.1\frac{bits/s}{Hz}$ para $\frac{E_b}{N_0} = 5.5dB$ e a capacidade soma tem valor $C_{soma} \approx 1.43\frac{bits/s}{Hz}$, resultando em um *gap* de $0.32\,\frac{bits/s}{Hz}$.

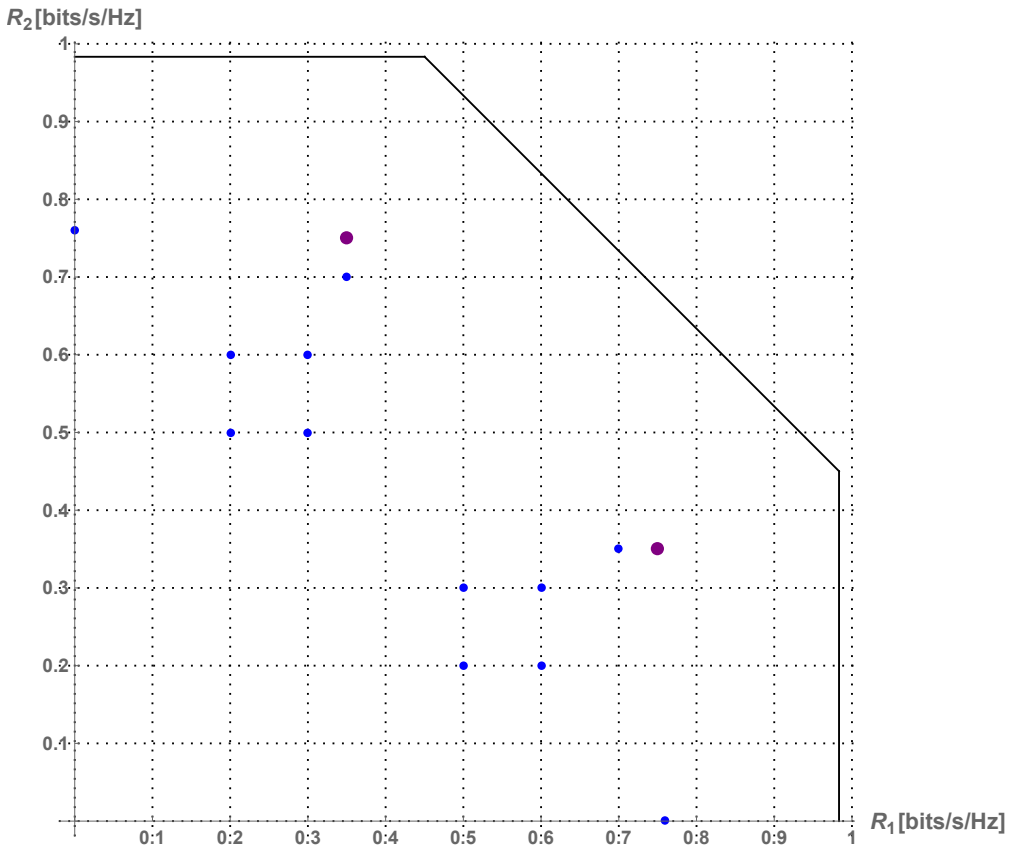

Figura 14 – Região de Capacidade calculada para 5.5 dB e pontos onde os pares de taxas resultaram em comunicação confiável (BER $< 10^{-5}$)

Note que já que o ponto $(0.75, 0.35)$ atinge a BER adequada no valor de $\frac{E_b}{N_0}$ objetivo usando a técnica SIC [4], todos os pontos no interior do retângulo $\{(0,0);(0.75,0);(0.75,0.35);(0,0.35)\}$ também são atingíveis. No entanto, à me-

dida que R_1 se aproxima de R_2, o desempenho do sistema cai. Como exemplo, o ponto $(0.4, 0.3)$ foi testado mas não obteve desempenho adequado. Acredita-se que isto ocorre pelo fato de que, ao aproximar a taxa dos dois usuários, a energia de símbolo das duas transmissões fica também mais próxima, intensificando o efeito de interferência entre os usuários.

Adicionalmente, foram realizados os cálculos para inferir qual valor de $\frac{E_b}{N_0}$ fornece a C_{soma} de $1.1 bits/s/Hz$. Isto ocorre para $\frac{E_b}{N_0} = 2.5 dB$. Pode-se concluir por esta medida que a distância entre este método e o limite teórico de Shannon é de 3 dB.

6 Conclusões

Foi construído um simulador de códigos polares e obtiveram-se resultados do desempenho deste código para o canal ponto a ponto e o canal de múltiplo acesso. Esta avaliação é uma contribuição para a literatura já que não foram encontrados resultados anteriores semelhantes.

Em relação às técnicas aplicadas, os resultados mostram que os benefícios do CRC só são alcançados quando deseja-se uma probabilidade de erro muito baixa (abaixo de 10^{-5}). Sua utilização deve ser avaliada caso a caso visto que requer tempo maior de simulação. O método de lista adaptativo utilizado em conjunto com CRC leva a diminuição no tempo de execução e pode ser interessante.

A utilização do decodificador em c++ diminui bastante o tempo de simulação. Para testes mais extensos e exigentes, recomenda-se escrever o simulador inteiramente em c++.

O nosso trabalho utilizou códigos polares em canais de múltiplo acesso e obteve distâncias de 3 dB e 0.32 bits/s/Hz do limite de Shannon. Pode-se dizer que os resultados foram bons, principalmente tomando como referência os trabalhos [17] e [18] onde a distância até o limite teórico ficou entre 1 e $2dB$ em ambos os trabalhos. Com a ressalva que ambos os trabalhos utilizaram códigos LDPC com tamanho de bloco 20000 ou maior. A proximidade do resultado aqui do limite de Shannon, comparada com a obtida em trabalhos anteriores, indica que estudos mais detalhados do uso de *Polar Codes* em canais de múltiplo acesso merecem uma investigação mais aprofundada.

6.1 Trabalhos Futuros

Os seguintes trabalhos futuros poderiam ser realizados:

- estudo mais aprofundado do uso de *Polar Codes* em canais de múltiplo acesso, variando parâmetros tais como tamanho da palavra código;
- utilização mais aprofundada da técnica SIC em paralelo e otimização do tempo de execução;
- estudo unificado comparando diferentes métodos de codificação para canais de múltiplo acesso tais como códigos polares, LDPC e *turbo codes*.

Referências

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, pp. 379–423 (Part 1); 623–656 (Part 2), Julho 1948. Citado 2 vezes nas páginas 15 e 21.

[2] E. Arikan, "Channel polarization: A method for constructing capacity achieving codes for symmetric binary-input memoryless channels.," *IEEE Transactions on Information Theory, vol. 55, no. 7*, p. 3051–3073, Julho 2009. Citado 10 vezes nas páginas 15, 22, 26, 27, 28, 29, 31, 39, 40 e 49.

[3] S. Lin and J. Daniel J. Costello, *Error Control Coding*. Upper Saddle River, NJ, US: Pearson Prentice Hall, 2 ed., 2004. Citado 4 vezes nas páginas 16, 21, 22 e 23.

[4] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley, 1991. Citado 8 vezes nas páginas 22, 23, 24, 43, 46, 47, 48 e 58.

[5] H. Vangala, E. Viterbo, and Y. Hong, "A comparative study of polar code constructions for the AWGN channel.," *arXiv:1501.02473 [cs.IT]*, 2015. http://arxiv.org/abs/1501.02473. Citado 3 vezes nas páginas 27, 31 e 40.

[6] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast Polar Decoders: Algorithm and Implementation," *IEEE Journal on Selected Areas in Communications*, vol. 32, pp. 946–957, Maio 2014. Citado na página 30.

[7] E. Arikan, "Systematic polar coding.," *IEE Communications Letters*, p. 15(8):860–862, Agosto 2011. Citado 5 vezes nas páginas 30, 32, 39, 55 e 56.

[8] I. Tal and A. Vardy, "List decoding of polar codes.," *arXiv:1206.0050 [cs.IT]*, 2012. Citado 5 vezes nas páginas 36, 38, 40, 41 e 49.

[9] "https://github.com/MashaYudi/PolarCodes." Citado na página 37.

[10] H. Vangala, E. Viterbo, and Y. Hong, "www.polarcodes.com." Citado 2 vezes nas páginas 38 e 51.

[11] http://ipgdemos.epfl.ch/polarcodes/files_php/main.php4. Citado 2 vezes nas páginas 39 e 40.

[12] E. Arikan, "Performance comparison of polar codes and Reed-Muller codes.," *IEEE Communications Letters, vol. 12, no. 6*, p. 447–449, Junho 2008. Citado 2 vezes nas páginas 39 e 40.

[13] H. Vangala, E. Viterbo, and Y. Hong, "Permuted successive cancellation decoder for polar codes.," *International Symposium on Information Theory and Applications (ISITA), Melbourne*, p. 438–442, Outubro 2014. Citado na página 40.

[14] B. Li, H. Shen, and D. Tse, "An Adaptive Successive Cancellation List Decoder for Polar Codes with Cyclic Redundancy Check.," *arXiv:1208.3091 [cs.IT]*, 2012. Citado 3 vezes nas páginas 41, 54 e 55.

[15] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg, "LLR-Based Successive Cancellation List Decoding of Polar Codes.," *arXiv:1401.3753v4 [cs.IT]*, Março 2015. Citado na página 42.

[16] J. Proakis, *Digital Communications*. Communications and signal processing, McGraw-Hill, 1995. Citado na página 45.

[17] N. Zheng, Y. He, B. Bai, A. M. So, and K. Yang, "Ldpc code design for gaussian multiple-access channels using dynamic exit chart analysis," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3679–3683, Março 2017. Citado 2 vezes nas páginas 48 e 60.

[18] Y. Chi, Y. Li, G. Song, and Y. Sun, "Partially repeated sc-ldpc codes for multiple-access channel," *IEEE Communications Letters*, vol. 20, pp. 1947–1950, Outubro 2016. Citado 2 vezes nas páginas 48 e 60.

[19] S. Önay, "Successive cancellation decoding of polar codes for the two-user binary-input mac," in *2013 IEEE International Symposium on Information Theory*, pp. 1122–1126, Julho 2013. Citado na página 48.

[20] K. Chen, K. Niu, and J. Lin, "Improved successive cancellation decoding of polar codes," *IEEE Transactions on Communications*, vol. 61, pp. 3100–3107, Agosto 2013. Citado na página 55.

Anexos

ANEXO A – Códigos utilizados

As páginas seguintes contém os códigos e scripts '.m' e '.cpp' utilizados neste trabalho.

1/18/20 7:45 PM C:\PolarCodes\Nova p...\bercalclistCRC.m 1 of 1

```
function [BER1,WER1,BER2,WER2] = bercalclistCRC(n,k1,k2,p,loop,Lmax,CRCsz,mode,
design)
BER1 = 0;
WER1 = 0;
BER2 = 0;
WER2 = 0;
%% construcao do codigo polar
[froze2, G] = pccfinal(n,k2,design);
[froze1, ~] = pccfinal(n,k1,design);
%% loops para simulacao Monte Carlo
for index = 1 : loop
    [ber1,wer1,ber2,wer2] = transimlistCRC(n,k1,k2,p,Lmax,CRCsz,froze2,froze1,mode,
G);
    %soma quantidade de palavras errados na transmissao e das taxas de erro
    %de bit
    WER1 = WER1 + wer1;
    WER2 = WER2 + wer2;
    BER1 = BER1 + ber1;
    BER2 = BER2 + ber2;
    %condicao de parada sendo 100 erros de palavra para ambos usuarios
    if ((WER1 > 100) && (WER2 > 100))
        break;
    end
end
clear G;
clear froze;
%% normaliza quantidade de erros para obter uma taxa ao inves de um valor
BER1 = BER1/(index);
WER1 = WER1/(index);
BER2 = BER2/(index);
WER2 = WER2/(index);
end
```

1/18/20 7:24 PM C:\PolarCodes\...\bermontecarlolistCRC.m 1 of 2

```
function [ebn0x, BERsys1, WERsys1, BERsys2, WERsys2] = bermontecarlolistCRC(n,rates1,
rates2,dBmin,points,dBmax,loop,Lmax,CRCsz,mode,design)
%% inicio
kk1 = floor(n*rates1);
kk2 = floor(n*rates2);
kk1 = kk1+CRCsz;    % Esse k eh o que o polar codes "ve", quando "k = bits de info"
for ser usado, subtrai-se CRCsz de k
kk2 = kk2+CRCsz;
pmin = (dBmax/10);
pmax = (dBmin/10);
oopp = length(kk1);
tic
ebn0x = fliplr(logspace(pmin,pmax,points));
BERsys1 = zeros(points,oopp);
WERsys1 = zeros(points,oopp);
BERsys2 = zeros(points,oopp);
WERsys2 = zeros(points,oopp);
%% loop
for index = 1:points
    for index2 = 1:oopp
        [BERsys1(index,index2),WERsys1(index,index2),BERsys2(index,index2),WERsys2
(index,index2)] = bercalclistCRC(n,kk1(index2),kk2(index2),ebn0x(index),loop,Lmax,
CRCsz,mode,design);
    end
end
%% processamento dos resultados e plot
ebn0x = log10(ebn0x).*10;
save('resultado','ebn0x','BERsys1','WERsys1','BERsys2','WERsys2');
toc

figure();
for w = 1:(oopp)
    semilogy(ebn0x,BERsys1(:,w),'-bo');

    hold on
end
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER (x1)');

figure;
for w = 1:(oopp)
    semilogy(ebn0x,WERsys1(:,w),'-bo');

    hold on
end
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('WER (x1)');
figure();
for w = 1:(oopp)
    semilogy(ebn0x,BERsys2(:,w),'-mo');

    hold on
end
```

```
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER (x2)');

figure;
for w = 1:(oopp)
    semilogy(ebn0x,WERsys2(:,w),'-mo');

    hold on
end
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('WER (x2)');
end
```

1/18/20 7:51 PM C:\PolarCodes\Nova pasta\...\buildword.m 1 of 1

```
function x = buildword(info,froze,n,k)
%% constroi a palavra encaixando os bits de informacao em seus lugares
if length(info) == k
    u = zeros(1,n);
    pos = 1;
    for j = 1:n
        if froze(j) == 1
            u(j) = info (pos);
            pos = pos + 1;
        else
            u(j) = 0;
        end
    end
    x = u;
else
    'ERROR: argument doesnt have the right ammount of information bits = '
    k
    x = NaN;
end
end
```

1/18/20 7:57 PM C:\PolarCodes\Nova...\calcw0w1primeiro.m 1 of 1

```
function [w0,w1] = calcw0w1primeiro(x,ebn0,k,centro1,centro2)
%% Estima as probabilidades de bit assumindo que dois usuarios dividem o canal
if isrow(x)
    x = x';
end
n = length(x);
w0 = 0.5*(normpdf(x,centro1+centro2,1)+normpdf(x,centro1-centro2,1));
w1 = 0.5*(normpdf(x,-1*(centro1+centro2),1)+normpdf(x,-1*(centro1-centro2),1));
end
```

1/18/20 8:03 PM C:\PolarCodes\Nova pa...\calcw0w1teste.m 1 of 1

```
function [w0,w1] = calcw0w1teste(x,ebn0,k,centro)
%% estima a probabilidade de bit para o canal com um so usuario
if isrow(x)
    x = x';
end
n = length(x);
w0 = (normpdf(x,centro,1));
w1 = (normpdf(x,-centro,1));
end
```

1/18/20 7:52 PM C:\PolarCodes\Nova pasta\Task...\calcz.m 1 of 1

```
function z = calcz(k,j,R,design)
%% calculo recursivo dos parametros de Bhattacharyya
if k == 1
    s = 10^(design/10);
    z = exp(-s);
else
    if j > (k/2)
        z = calcz(k/2,j-(k/2),R,design);
        z = z*z;
    else
        z = calcz(k/2,j,R,design);
        z = 2*z-z*z;
    end
end
end
```

```
//c++ do decodificador com as classes, modificado
//original feito por MashaYudi
#include "decoder.h"
#define _USE_MATH_DEFINES
#include <math.h>
#include "mex.h"
using namespace std;
decoder::decoder(int _L, int _n, set<int> _u) {
    L = _L;
    n = _n;
    m = log2(n);
    u = _u;
    //dispersion = _dispersion;
    inactivePathIndices.reserve(L);
    activePath.resize(L);

    arrayPointer_P = new long double***[m + 1];
    for (s = 0; s < m + 1; s++)
        arrayPointer_P[s] = new long double**[L];

    arrayPointer_C = new int***[m + 1];
    for (s = 0; s < m + 1; s++)
        arrayPointer_C[s] = new int**[L];

    pathIndexToArrayIndex = new int*[m + 1];
    for (s = 0; s < m + 1; s++)
        pathIndexToArrayIndex[s] = new int[L];

    row.reserve(L);
    for (s = 0; s < m + 1; s++) {
        inactiveArrayIndices.push_back(row);
    }

    arrayReferenceCount = new int*[m + 1];
    for (s = 0; s< m + 1; s++)
        arrayReferenceCount[s] = new int[L];

    for (lambda = 0; lambda <= m; lambda++) {
        for (s = 0; s < L; s++) {
            arrayPointer_P[lambda][s] = new long double*[(int)pow(2, m - lambda)];
            //cout << "(int)pow(2, m - lambda) = " << (int)pow(2, m - lambda) << ↙
endl;
            arrayPointer_C[lambda][s] = new int*[(int)pow(2, m - lambda)];
            for (int i = 0; i < (int)pow(2, m - lambda); i++) {
                arrayPointer_P[lambda][s][i] = new long double[2];
                arrayPointer_C[lambda][s][i] = new int[2];
            }
            //arrayReferenceCount[lambda][s] = 0;
            //inactiveArrayIndices[lambda].push_back(s);
        }
    }
    probForks = new long double*[L];
    for (l = 0; l< L; l++)
        probForks[l] = new long double[2];
```

```
    contForks = new bool*[L];
    for (l = 0;l < L; l++)
        contForks[l] = new bool[2];

    temp_cont.resize(2 * L);

    c = new int*[n];
    for (s = 0; s < n; s++)
        c[s] = new int[L];
}
decoder::~decoder() {
    for (s = 0; s < inactiveArrayIndices.size(); s++){
        inactiveArrayIndices.at(s).clear();
        inactiveArrayIndices.at(s).shrink_to_fit();
    }
    inactiveArrayIndices.clear();
    inactiveArrayIndices.shrink_to_fit();
    activePath.clear();
    activePath.shrink_to_fit();

    for (s = 0; s < m + 1; s++) {
        delete[] pathIndexToArrayIndex[s];
    }
    delete[] pathIndexToArrayIndex;

    for (lambda = 0; lambda < m + 1; lambda++) {
        for (s = 0; s < L; s++) {
            for (int i = 0; i < (int)pow(2, m - lambda); i++) {
                delete[] arrayPointer_P[lambda][s][i];
                delete[] arrayPointer_C[lambda][s][i];
            }
            delete[] arrayPointer_P[lambda][s];
            delete[] arrayPointer_C[lambda][s];

        }
        delete[] arrayPointer_P[lambda];
        delete[] arrayPointer_C[lambda];
        delete[] arrayReferenceCount[lambda];
    }
    delete[] arrayReferenceCount;

    delete[] arrayPointer_P;

    delete[] arrayPointer_C;
    inactivePathIndices.clear();
    inactivePathIndices.shrink_to_fit();
    for (s = 0; s < n; s++) {
        delete[] c[s];
    }
    delete[] c;
    temp_cont.clear();
    temp_cont.shrink_to_fit();
    delete[] probForks;
    delete[] contForks;
```

```
    row.clear();
    row.shrink_to_fit();
}

void decoder::initialization() {

    for (lambda = 0; lambda <= m; lambda++) {
        for (s = 0; s < L; s++) {
            arrayReferenceCount[lambda][s] = 0;
            inactiveArrayIndices[lambda].push_back(s);
        }
    }
    for (l = 0; l < L; l++) {
        activePath[l] = false;
        inactivePathIndices.push_back(l);
    }

}

int decoder::assignInitialPath() {
    l = inactivePathIndices[inactivePathIndices.size() - 1];
    inactivePathIndices.pop_back();
    activePath[l] = true;
    for (lambda = 0; lambda <= m; lambda++) {
        s = inactiveArrayIndices[lambda][inactiveArrayIndices[lambda].size() - 1];
        inactiveArrayIndices[lambda].pop_back();
        pathIndexToArrayIndex[lambda][l] = s;

        arrayReferenceCount[lambda][s] = 1;
    }
    return l;
}

long double** decoder::getArrayPointer_P(int lambda, int l) {
    s = pathIndexToArrayIndex[lambda][l];
    if (arrayReferenceCount[lambda][s] == 1) {
        s2 = s;
    }
    else {
        s2 = inactiveArrayIndices[lambda][inactiveArrayIndices[lambda].size() - 1];
        inactiveArrayIndices[lambda].pop_back();
        for (int i = 0; i < (int)pow(2, m - lambda); i++) { //(int)pow(2, m - lambda↙
(+1 or not));
            arrayPointer_P[lambda][s2][i][0] = arrayPointer_P[lambda][s][i][0];
            arrayPointer_P[lambda][s2][i][1] = arrayPointer_P[lambda][s][i][1];

            arrayPointer_C[lambda][s2][i][0] = arrayPointer_C[lambda][s][i][0];
            arrayPointer_C[lambda][s2][i][1] = arrayPointer_C[lambda][s][i][1];
        }

        arrayReferenceCount[lambda][s]--;
        arrayReferenceCount[lambda][s2] = 1;
        pathIndexToArrayIndex[lambda][l] = s2;
    }
    return arrayPointer_P[lambda][s2];
```

```
}
int** decoder::getArrayPointer_C(int lambda, int l) {
    s = pathIndexToArrayIndex[lambda][l];
    if (arrayReferenceCount[lambda][s] == 1) {
        s2 = s;
    }
    else {
        s2 = inactiveArrayIndices[lambda][inactiveArrayIndices[lambda].size() - 1];
        inactiveArrayIndices[lambda].pop_back();
        for (int i = 0; i < (int)pow(2, m - lambda); i++) {
            arrayPointer_P[lambda][s2][i][0] = arrayPointer_P[lambda][s][i][0];
            arrayPointer_P[lambda][s2][i][1] = arrayPointer_P[lambda][s][i][1];

            arrayPointer_C[lambda][s2][i][0] = arrayPointer_C[lambda][s][i][0];
            arrayPointer_C[lambda][s2][i][1] = arrayPointer_C[lambda][s][i][1];
        }
        arrayReferenceCount[lambda][s]--;
        arrayReferenceCount[lambda][s2] = 1;
        pathIndexToArrayIndex[lambda][l] = s2;
    }
    return arrayPointer_C[lambda][s2];
}
void decoder::recursivelyCalcP(int lambda, int phi) {
    long double **P_lambda, **P_prelambda;
    int **C_lambda;
    if (lambda == 0) return;
    psi = phi / 2;
    if (phi % 2 == 0)
        recursivelyCalcP(lambda - 1, psi);
    long double sigma2 = 0;
    for (l = 0; l < L; l++) {
        if (activePath[l] == false)
            continue;
        P_lambda = getArrayPointer_P(lambda, l);
        P_prelambda = getArrayPointer_P(lambda - 1, l);
        C_lambda = getArrayPointer_C(lambda, l);
        int tml = pow(2, m - lambda);
        for (beta = 0; beta < tml; beta++) {
            if (phi % 2 == 0) {
                //P_lambda[beta][0] = (P_prelambda[2*beta][0] * P_prelambda[2*beta + ↙
1][0] + P_prelambda[2*beta][1] * P_prelambda[2*beta + 1][1]) / 2;
                P_lambda[beta][0] = (P_prelambda[beta][0] * P_prelambda[beta + tml] ↙
[0] + P_prelambda[beta][1] * P_prelambda[beta + tml][1]) / 2;
                sigma2 = sigma2 > P_lambda[beta][0] ? sigma2 : P_lambda[beta][0];

                //P_lambda[beta][1] = (P_prelambda[2*beta][1] * P_prelambda[2*beta + ↙
1][0] + P_prelambda[2*beta][0] * P_prelambda[2*beta + 1][1]) / 2;
                P_lambda[beta][1] = (P_prelambda[beta][1] * P_prelambda[beta + tml] ↙
[0] + P_prelambda[beta][0] * P_prelambda[beta + tml][1]) / 2;
                sigma2 = sigma2 > P_lambda[beta][1] ? sigma2 : P_lambda[beta][1];
            }
            else {
                u2 = C_lambda[beta][0];
                //P_lambda[beta][0] = P_prelambda[2 * beta][u] * P_prelambda[2 * beta ↙
+ 1][0] / 2;
```

```
                P_lambda[beta][0] = P_prelambda[beta][u2] * P_prelambda[beta + tml] ↙
[0] / 2;
                sigma2 = sigma2 > P_lambda[beta][0] ? sigma2 : P_lambda[beta][0];

                //P_lambda[beta][1] = P_prelambda[2 * beta][u2 ^ 1] * P_prelambda[2 * ↙
beta + 1][1] / 2;
                P_lambda[beta][1] = P_prelambda[beta][u2 ^ 1] * P_prelambda[beta + ↙
tml][1] / 2;
                sigma2 = sigma2 > P_lambda[beta][1] ? sigma2 : P_lambda[beta][1];
            }
        }
    }
    //sigma2 = 1;
    for (l = 0; l < L; l++) {
        if (activePath[l] == false)
            continue;
        P_lambda = getArrayPointer_P(lambda, l);
        for (beta = 0; beta < pow(2, m - lambda); beta++) {
            P_lambda[beta][0] = P_lambda[beta][0] / sigma2;
            P_lambda[beta][1] = P_lambda[beta][1] / sigma2;
        }
    }

}

void decoder::continuePaths_UnfrozenBit(int phi) {
    //long double **probForks = new long double*[L];
    //for (int count = 0; count < L; count++)
    //  probForks[count] = new long double[2];

    int i = 0;
    //
    for (l = 0; l < L; l++) {
        if (activePath[l] == true) {
            long double **P_m = getArrayPointer_P(m, l);
            probForks[l][0] = P_m[0][0];
            probForks[l][1] = P_m[0][1];
            i++;
        }
        else {
            probForks[l][0] = -1;
            probForks[l][1] = -1;
        }
    }
    //bool **contForks = new bool*[L];
//  for (int count = 0; count < L; count++)
    //  contForks[count] = new bool[2];
    rho = (2 * i <= L) ? 2 * i : L;// min(2 * i, L);
    i = 0;
    //vector<long double> temp_cont;
    //temp_cont.resize(2 * L);
    //long double* temp = new long double[2 * L];
    for (h = 0; h < L; h++) {
        for (j = 0; j < 2; j++) {
            temp_cont[i] = probForks[h][j];
```

```
            i++;
        }
    }
/*
    for (int j = 0; j < 2; j++) {
    for (int h = 0; h < L; h++) {

            cout << probForks[h][j] << " ";
        }
        cout << endl;
    }*/
    Sort(temp_cont, 0, i - 1);
    def = temp_cont[2 * L - rho];
    i = 0;
    for (h = 0; h < L; h++) {
        for (j = 0; j < 2; j++) {
            if ((probForks[h][j] > def) && (i < rho)) {   //add stoppin' condition
                contForks[h][j] = true;
                i++;
            }
            else
                contForks[h][j] = false;
        }
    }
    //in still < rho, look for some equal
    for (h = 0; h < L; h++) {
        for (j = 0; j < 2; j++) {
            if (i < rho) {
                if (probForks[h][j] == def) {
                    contForks[h][j] = true;
                    i++;
                }
            }
        }
    }
    /*for (int j = 0; j < 2; j++) {
    for (int h = 0; h < L; h++) {

            cout << contForks[h][j] << " ";
        }
        cout << endl;
    }
    */
    for (l = 0; l < L; l++) {
        if (activePath[l] == false)
            continue;
        if ((contForks[l][0] == false) && (contForks[l][1] == false))
            killPath(l);
    }
    for (l = 0; l < L; l++) {
        if ((contForks[l][0] == false) && (contForks[l][1] == false))
            continue;
        int** C_m = getArrayPointer_C(m, l);
        if ((contForks[l][0] == true) && (contForks[l][1] == true)) {
```

```
            C_m[0][phi % 2] = 0;
            l2 = clonePath(l);
            C_m = getArrayPointer_C(m, l2);
            C_m[0][phi % 2] = 1;
        }
        //exactly one fork is good
        else {
            if (contForks[l][0] == true) {
                C_m[0][phi % 2] = 0;
            }
            else
                C_m[0][phi % 2] = 1;
        }
    }
}

int decoder::clonePath(int l) {
    l2 = inactivePathIndices[inactivePathIndices.size() - 1];
    inactivePathIndices.pop_back();
    activePath[l2] = true;

    for (lambda = 0; lambda <= m; lambda++) {
        s = pathIndexToArrayIndex[lambda][l];
        pathIndexToArrayIndex[lambda][l2] = s;
        arrayReferenceCount[lambda][s]++;
    }
    return l2;
}
void decoder::killPath(int l) {

    activePath[l] = false;
    inactivePathIndices.push_back(l);
    for (lambda = 0; lambda <= m; lambda++) {
        s = pathIndexToArrayIndex[lambda][l];
        arrayReferenceCount[lambda][s]--;
        if (arrayReferenceCount[lambda][s] == 0)
            inactiveArrayIndices[lambda].push_back(s);
    }
}
int decoder::Partition(vector<long double>& vector, int start, int end) {

    s = start - 1;
    long double elem = vector[end];
    //exchange [start + index] with [end]
    for (j = start; j < end; j++) {
        if (vector[j] <= elem) {
            //exchange [j] and [i]
            s++;
            swap(vector[j], vector[s]);
        }
    }
    //exchange [i + 1] and [end] and exchange [ i + 1 + .size()/2] and [end + .size() ↙
/2]
    swap(vector[s + 1], vector[end]);
    return s + 1;
```

```
}

void decoder::Sort(vector<long double>& vector, int start, int end) {
    if (start < end) {
        h = Partition(vector, start, end);
        Sort(vector, start, h - 1);
        Sort(vector, h + 1, end);
    }
}

void decoder::recursivelyUpdateC(int lambda, int phi) {
    psi = phi / 2;
    for (l = 0; l < L; l++) {
        if (activePath[l] == false)
            continue;
        int** C_lambda = getArrayPointer_C(lambda, l);
        int** C_prelambda = getArrayPointer_C(lambda - 1, l);
        h = pow(2, m - lambda);
        for (beta = 0; beta < h; beta++) {
            C_prelambda[beta][psi % 2] = C_lambda[beta][0] ^ C_lambda[beta][1];
            C_prelambda[beta + h][psi % 2] = C_lambda[beta][1];
        }
    }
    if (psi % 2 == 1)
        recursivelyUpdateC(lambda - 1, psi);

}
int** decoder::decode(double* w0, double* w1) {
    initialization();
    int po,popo,p2,L3;
    int* Lgood;
    Lgood = new int [L];
    int** C_m;
    int** C_0;
    long double** P_m;
    l = assignInitialPath();
    long double** P_0 = getArrayPointer_P(0, l);

    for (beta = 0; beta < n; beta++) {
        //double L = 2 * y[beta] / dispersion;

        //P_0[beta][1] = 1. / (exp(L) + 1);
        //P_0[beta][0] = 1. - P_0[beta][1];

        P_0[beta][0] = w0[beta];

        P_0[beta][1] = w1[beta];
    }

    for (phi = 0; phi < n; phi++) {
        recursivelyCalcP(m, phi);

        if (u.find(phi) != u.end()/*u[phi] == false*/) {
            for (l = 0; l < L; l++) {
                if (activePath[l] == false)
```

```
                continue;
            C_m = getArrayPointer_C(m, l);
            C_m[0][phi % 2] = 0; // zero or not zero? or what?!
        }

    }
    else {
        continuePaths_UnfrozenBit(phi);
    }

    if (phi % 2 == 1)
        recursivelyUpdateC(m, phi);

}

l2 = 0;
p = 0;

for (l = 0; l < L; l++) {
    if (activePath[l] == false)
        continue;
    C_m = getArrayPointer_C(m, l);
    P_m = getArrayPointer_P(m, l);

    if (p < P_m[0][C_m[0][1]]) {
        l2 = l;
        p = P_m[0][C_m[0][1]];
    }
}
//int** C_0 = getArrayPointer_C(0, l2);
//int** c = new int[n][L];
po = 0;
Lgood[0] = l2;
p = 0;
for (L3 = 1; L3 < L; L3++) {
    p = 0;
    for (l = 0; l < L; l++) {
        for (popo = 0; popo < L; popo++){
            if (l == Lgood[popo])
                po = 1;
        }
        if (po == 1){
            po = 0;
            continue;
        }
        if (activePath[l] == false)
            continue;
        C_m = getArrayPointer_C(m, l);
        P_m = getArrayPointer_P(m, l);

        if (p < P_m[0][C_m[0][1]]) {
            l2 = l;
            p = P_m[0][C_m[0][1]];
        }
    }
```

```
        Lgood[L3] = l2;
    }
    /*
    for (l = 0; l < L; l++) {
        mexPrintf("%d\n",Lgood[l]);
        mexEvalString("drawnow;");
    }
    */
    //for (s = 0; s < n; s++)
    //  c[s][0] = C_0[s][0];
    //po = 0;
    for (l = 0; l < L; l++) {
        C_0 = getArrayPointer_C(0, Lgood[l]);
        for (s = 0; s < n; s++)
            c[s][l] = C_0[s][0];
        //po++;
    }

    delete[] Lgood;
    return c;
}

void decoder::check() {

    /*cout << "arryReferenceCount content: " << endl;
    for (int lambda = 0; lambda < m + 1; lambda++) {
        for (int s = 0; s < L; s++) {
            cout << "arrayReferenceCount[" << lambda << "][" << s << "]: " << ↙
arrayReferenceCount[lambda][s] << endl;
        }
        cout << "--------" << endl;
    }*/
    /*
    cout << "inactivePathIndices content (stack):" << endl;
    for (int i = 0; i < inactivePathIndices.size(); i++)
        cout << inactivePathIndices[i] << " ";
    cout << endl;


    cout << "pathIndexToArrayIndex content:" << endl;
    for (int lambda = 0; lambda < m + 1; lambda++) {
        for (int s = 0; s < L; s++)
            cout << "pathIndexToArrayIndex[" << lambda << "][" << s << " ] is " << ↙
pathIndexToArrayIndex[lambda][s] << endl;
    }
    cout << "------" << endl;

    cout << "activePath" << endl;
    for (int a = 0; a < L; a++) {
        cout << activePath[a] << " ";
    }
    cout << endl;
    */
    cout << "Array Pointer P content: " << endl;
    for (int lambda = 0; lambda < m + 1; lambda++) {
```

```
        for (int s = 0; s < L; s++) {
            cout << "arrayPointer_P[ " << lambda << " ][ " << s << " ]" << endl;
            for (int i = 0; i < (int)pow(2.0, m - lambda); i++) {
                cout << arrayPointer_P[lambda][s][i][0] << " ";
                //cout << endl;
                //for (int i = 0; i < (int)pow(2.0, m - lambda); i++)
                cout << arrayPointer_P[lambda][s][i][1] << " ";
                cout << endl;
            }
            cout << endl;
            cout << "--------" << endl;
        }
    }


    cout << "Array Pointer C content: " << endl;
    for (int lambda = 0; lambda < m + 1; lambda++) {
        for (int s = 0; s < L; s++) {
            cout << "arrayPointer_C[ " << lambda << " ][ " << s << " ]" << endl;
            int** te = arrayPointer_C[lambda][s];
            for (int i = 0; i < (int)pow(2.0, m - lambda); i++) {
                cout << te[i][0] << " ";
                //cout << arrayPointer_C[lambda][s][i][0] << " ";
                //cout << endl;
                //for (int i = 0; i < (int)pow(2.0, m - lambda); i++)
                cout << te[i][1] << " ";
                //cout << arrayPointer_C[lambda][s][i][1] << " ";
                cout << endl;
            }
            cout << endl;
            cout << "--------" << endl;
        }
    }
}
```

1/18/20 7:50 PM C:\PolarCodes...\encodepolarnorevteste.m 1 of 1

```
function tx = encodepolarnorevteste(info,froze,n,k,G)
%% codificacao codigos polares por multiplicacao matricial
info = buildword(info,froze,n,k);
if iscolumn(info)
    info = info';
end
tx = info*G;
tx = mod(tx,2)';
end
```

1/18/20 8:01 PM C:\PolarCodes\Nov...\maindecoder4CRC.cpp 1 of 3

```
// decodificador em c++ contendo interface matlab c++
#include <vector>
#include <iostream>
#include <set>
#include <fstream>
#include <functional>
#include "decoder.h"
#include "mex.h"
#include <chrono>
#include <random>
#include <math.h>
#include <queue>
using namespace std;
void mexFunction(
                 int            nlhs,
                 mxArray       *plhs[],
                 int            nrhs,
                 const mxArray *prhs[] )
{
    //dispersion = E_s * n / (2 * k* ebn0);
    //mexPrintf("static destructor\n");
    //mexEvalString("drawnow;");
    double *L0;
    double *n0;
    double *k0;
    double *frozen0;
    double *w0;
    double *w1;
    //long double dispersion;
    //double *ebn00;
    int **decodeout;
    int frozen1;
    int L;
    int n;
    int k;
    set<int> frozen;
    set<int>::iterator iterador = frozen.begin();
    float ebn0;
    //long double *info;
    double *output;
    int theta = 1024;
    int it;
    int it2;
    if (nrhs ==  6) {
        L0 = mxGetPr(prhs[0]);
        n0 = mxGetPr(prhs[1]);
        k0 = mxGetPr(prhs[2]);
        frozen0 = mxGetPr(prhs[3]);
        //ebn00 = mxGetPr(prhs[3]);
        w0 = mxGetPr(prhs[4]);
        w1 = mxGetPr(prhs[5]);
    }else{
        mexErrMsgTxt("Usage: [output] = maindecoder(L, n, k, frozen, ebn0, info)");
    }
    if (nlhs !=  1) {
```

```
        mexErrMsgTxt("Usage: [output] = maindecoder(L, n, k, frozen, ebn0, info)");
    }
    L = (int)L0[0];
    n = (int)n0[0];
    k = (int)k0[0];

    for(it = 0; it < (n-k); it ++){
        frozen1 = (int)frozen0[it];
        frozen.insert(iterador,frozen1);
        ++iterador;
        //frozen->insert((int)frozen0[n-1-it]);
        //frozen[it] = (int)frozen0[it];
    }

    //frozen = (int*)frozen0;
    //ebn0 = (float)ebn00[0];
    //double *P_j = new double[n];
    /*
    for(it = 0; it < n; it ++){
        info[it] = (long double)info0[it];
    }
     */
    //info = (long double*)info0;
    //dispersion = n / (2 * k* ebn0);
    //meanCount(n, dispersion, P_j, n - k, &frozen);
    /*
    for(iterador=frozen.begin(); iterador!=frozen.end();++iterador){
    mexPrintf("%d\n",*iterador);
    mexEvalString("drawnow;");
    //ii+=1;
    //cout<<ii<<endl;
    }
    */
    decoder decodex(L, n, frozen);
    //mexPrintf("static destructor\n");
    //mexEvalString("drawnow;");
    plhs[0] = mxCreateDoubleMatrix(n*L, 1, mxREAL );
    //plhs[0] = mxCreateNumericMatrix(n, L, mxINT8_CLASS, mxREAL );
    output = mxGetPr(plhs[0]);

    decodeout = decodex.decode(w0,w1);

    //output = decodeout;
    /*
    for(it = 0; it < n; it ++){
        mexPrintf("%d\n",decodeout[it]);
        mexEvalString("drawnow;");
    }
     */
    for (it2 = 0; it2 < L; it2 ++){
        for(it = 0; it < n; it ++){
            output[it+(n*it2)] = (double)decodeout[it][it2];
            //output[it][it2] = decodeout[it][it2];
        }
    }
```

```
    frozen.clear();
    /*
    delete[] L0;
    delete[] n0;
    delete[] k0;
    delete[] frozen0;
    delete[] w0;
    delete[] w1;
    //long double dispersion;
    delete[] decodeout;
    //output = (double*)decoder.decode(info);
     */
}
```

1/18/20 7:48 PM C:\PolarCodes\Nova pasta\Ta...\makeCRC.m 1 of 1

```
function y = makeCRC(x,size)
%% aplica o CRC na palavra
tam = length(x);
if isrow(x)
    x = x';
end
y2 = zeros(tam+size,1);
y2(1:tam) = x;
CRCkey = zeros(size+1,1);
for i = 1:4:(size)
    CRCkey(i) = 1;
    CRCkey(i+1) = 1;
end
for i = 1:tam
    if y2(i) == 1
        y2(i:i+size) = mod(y2(i:i+size)+CRCkey,2);
    end
    if sum(y2(1:tam) ~= 0) == 0
        i = tam+2;
    end
end
y = [x;y2(tam+1:tam+size)];
end
```

1/18/20 7:34 PM C:\PolarCodes\Nova pasta\T...\pccfinal.m 1 of 1

```
function [froze,G] = pccfinal (n,k,design)
%construcao do codigo polar
z = zeros(n,1);
out = zeros(n,1);
%% obtencao dos parametros de Bhattacharyya
for i = 1:n
    z(i) = calcz(n,i,k/n,design);
end
[~,order] = sort(z);
%% ordenacao dos parametros de Bhattacharyya e escolha dos frozen bits
for j = 1:k
    out(order(j)) = 1;
end
froze = logical(out);
froze = bitrevorder(froze);
%% construcao da matriz de codificacao
F = sparse([1 0;1 1]);
G = sparse(1);
for i = 1:log2(n)
    G = kron(F,G);
end
end
```

1/18/20 7:49 PM C:\PolarCodes\Nova pasta\Ta...\sencode.m 1 of 1

```
function x = sencode(u,froze,n,k,G)
%% codificacao sistematica
x = encodepolarnorevteste(u,froze,n,k,G);
x = takeinfo(x,froze,n,k);
x = encodepolarnorevteste(x,froze,n,k,G);
end
```

```
function out = sencodeCRC(info,froze,n,k,CRCsz,G)
%% codificacao sistematica com CRC
if CRCsz > 0
info = makeCRC(info,CRCsz);
end
out = sencode(info,froze,n,k,G);
end
```

1/18/20 7:54 PM C:\PolarCodes\Nova pasta\T...\takeinfo.m 1 of 1

```
function x = takeinfo (y,froze,n,k)
%% obtem os bits de informacao de uma palavra formada por buildword.m
u = zeros(k,1);
pos = 1;
for j = 1:n
    if froze(j) == 1
        u(pos) = y(j);
        pos = pos + 1;
    end
end
x = u;
end
```

1/18/20 7:46 PM C:\PolarCodes\Nova p...\transimlistCRC.m 1 of 1

```
function [BER1,WER1, BER2, WER2] = transimlistCRC(n,k1,k2,ebn0,Lmax,CRCsz,froze2, ↙
froze1,mode,G)
%simulacao de uma transmissao
%% geracao, codificacao e modulacao da informacao
info1 = randi([0 1],k1-CRCsz,1);
info2 = randi([0 1],k2-CRCsz,1);
centro1 = sqrt(2*ebn0*((k1-CRCsz)/n));
centro2 = sqrt(2*ebn0*((k2-CRCsz)/n));
tx1 = sencodeCRC(info1,froze1,n,(k1),CRCsz,G);
tx1 = (2*tx1-1)*centro1;
tx2 = sencodeCRC(info2,froze2,n,(k2),CRCsz,G);
tx2 = (2*tx2-1)*centro2;
%% simulacao do canal
tx = tx1+tx2;
h = randn(n,1);
rx = tx+h;
%% obtencao probabilidade de bit e decodificacao
[w0,w1] = calcw0w1primeiro(rx,ebn0,(k1-CRCsz),centro1,centro2);
rx1 = truedecoder(w0,w1,Lmax,CRCsz,froze1,n,(k1),mode);
error1 = sum(info1 ~= rx1); % quantidade de bits errados
genie = sencodeCRC(rx1,froze1,n,(k1),CRCsz,G);
genie = (2*genie-1)*sqrt(2*ebn0*((k1-CRCsz)/n));
rx02 = rx - genie;
[w0,w1] = calcw0w1teste(rx02,ebn0,(k2-CRCsz),centro2);
rx2 = truedecoder(w0,w1,Lmax,CRCsz,froze2,n,(k2),mode);
error2 = sum(info2 ~= rx2); % quantidade de bits errados
WER1 = 0;
WER2 = 0;
%% verifica se houve erro de palavra
if error1 > 0
    WER1 = 1;
end
if error2 > 0
    WER2 = 1;
end
%% obtencao da taxa de erro de bit
BER1 = error1/(k1-CRCsz);
BER2 = error2/(k2-CRCsz);
end
```

1/18/20 7:58 PM C:\PolarCodes\Nova past...\truedecoder.m 1 of 1

```
function y = truedecoder(w0,w1,Lmax,CRCsz,froze,n,k,~)
%% decodificador MATLAB
frozen = zeros(n-k,1);
ii = 1;
for i = 1:n
    if froze(i) == 0
        frozen(ii) = (i-1);
        ii = ii+1;
    end
end
y = maindecoder5(Lmax, n, k, frozen, w0,w1); % decodificador c++
y = not(y);
y = takeinfo(y,froze,n,(k+CRCsz));
y = y(1:k);
end
```